SQLGenie: A Practical LLM based System for Reliable and Efficient SQL Generation

Pushpendu Ghosh
RBS Tech Sciences
Amazon
gpushpen@amazon.com

Aryan Jain
RBS Tech Sciences
Amazon
arynjn@amazon.com

Promod Yenigalla
RBS Tech Sciences
Amazon
promy@amazon.com

Abstract

Large Language Models (LLMs) enable natural language to SQL conversion, allowing users to query databases without SQL expertise. However, generating accurate, efficient queries is challenging due to ambiguous intent, domain knowledge requirements, and database constraints. Extensive reasoning improves SQL quality but increases computational costs and latency. We propose SQLGenie, a practical system for reliable SQL generation. It consists of three components: (1) **Table Onboarder**, which analyzes new tables, optimizes indexing, partitions data, identifies foreign key relationships, and stores schema details for SQL generation; (2) **SQL Generator**, an LLM-based system producing accurate SQL; and (3) **Feedback Augmentation**, which filters correct query-SQL pairs, leverages multiple LLM agents for complex SQL, and stores verified examples. SQLGenie achieves state-of-the-art performance on public benchmarks (92.8% execution accuracy on WikiSQL, 82.1% on Spider, 73.8% on BIRD) and internal datasets, surpassing the best single-LLM baseline by 21.5% and the strongest pipeline competitor by 5.3%. Its hybrid variant optimally balances accuracy and efficiency, reducing generation time by 64% compared to traditional multi-LLM approaches while maintaining competitive accuracy.

1 Introduction

Text-to-SQL generation has become a crucial capability in industry, enabling non-technical users to query databases using natural language. As organizations accumulate vast structured datasets, democratizing data access through natural language interfaces marks a significant advancement in enterprise analytics. However, developing robust text-to-SQL systems for production presents unique challenges, including handling domain-specific terminology, ensuring high accuracy across diverse schemas, and maintaining query performance at scale.

Large Language Models (LLMs) have demonstrated remarkable SQL generation capabilities, surpassing rule-based and supervised approaches by interpreting complex query intents and producing syntactically correct SQL with minimal explicit training. However, they remain unreliable in industrial applications, frequently hallucinating column names, misinterpreting intent, or generating logically incorrect queries. Common errors include faulty join conditions, improper aggregations, and mismatches between filter values and actual database content. While ensemble methods and multi-agent approaches improve accuracy by splitting SQL generation into planning and execution phases, they require multiple LLM calls, increasing latency and computational costs—making them impractical for real-time production use.

To address these challenges, we propose SQLGenie, a practical and efficient SQL generation framework that integrates an agentic approach with historical query reuse. SQLGenie incorporates a structured table onboarding process to capture essential database characteristics, a flexible SQL generation pipeline that leverages verified examples when available, and a feedback-driven augmentation mechanism for continuous improvement. By balancing accuracy, efficiency, and adaptability to domain-specific requirements, SQLGenie advances the state of the art in industrial text-to-SQL systems.

2 Related Works

Text-to-SQL systems have evolved from rule-based approaches to neural architectures. Early methods relied on handcrafted rules and templates, requiring extensive human engineering to map natural language queries to SQL (Hendrix et al., 1978). While pioneering, these approaches lacked scalability across domains.

The advent of deep learning introduced encoder-

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 1004–1012

decoder architectures for direct translation of text to SQL. Seq2SQL (Zhong et al., 2017a) leveraged reinforcement learning to enhance accuracy, while attention mechanisms improved query and schema alignment (Bahdanau et al., 2016). Pretrained models like BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019), fine-tuned on datasets such as Spider (Yu et al., 2019a), set new benchmarks but struggled with complex SQL and cross-domain generalization. The emergence of LLMs like GPT-4 (OpenAI, 2024) marked a shift, significantly improving SQL generation with minimal human intervention.

Recent research enhances LLM performance via multi-agent pipelines. For instance, MAC-SQL [(Wang et al., 2025) introduces a multi-agent framework with agents for schema linking, question decomposition, and iterative SQL generation and refinement. Another work presents MageSQL (Shen et al., 2024), a system that orchestrates multiple agents in a pipeline, allowing users to customize prompts and agent functionalities for enhanced Text-to-SQL performance. Retrieval-Augmented Generation (RAG) (Lewis et al., 2021) integrates retrieval mechanisms to incorporate relevant context, reducing hallucinations. Actor-critic frameworks iteratively refine SQL generation, while schema linking techniques in IRNet and RAT-SQL (Guo et al., 2019) align natural language with database structures, improving query precision. These advancements underscore the dynamic progress in Text-to-SQL, with ongoing efforts to optimize LLM-based approaches for precise database querying.

3 Methodolgy

3.1 Setup: Onboarding new tables

Given a relational database D consisting of n tables, each table T_i ($1 \leq i \leq n$) with c_i columns undergoes an onboarding process to enhance query performance and interoperability. Along with schema generation, the process includes: 1. Maintenance of column synonyms 2. Selection of optimal partitioning, indexing and primary keys, 3. Mapping of T_i's columns to relevant columns in previously onboarded tables $\{T_1, T_2, \ldots, T_{i-1}\}$ to define permissible joins, 4. Caching of frequent values for categorical columns to facilitate generation of accurate filters.

3.1.1 Column metadata

An LLM fine-tuned on internal knowledge base is used to generate synonyms and abbreviations (both expanded and shortened forms) for each column in the table. Additionally, each column is assigned an appropriate aggregation function, such as `STRING_AGG`, `SUM`, `MIN`, `MAX`, `COUNT` or `AVG`.

3.1.2 Column selection for partioning, indexing and PK-FK mapping

Partitioning Column Selection: To optimize query performance, we select partitioning columns based on high cardinality and frequent usage in query filters. Columns exhibiting a broad distribution of unique values, such as timestamps or region-based identifiers, are prioritized to ensure balanced partitions.

Indexing Column Selection: Indexing decisions are guided by selectivity and query workload characteristics. Columns frequently appearing in `JOIN`, `ORDER BY`, or `GROUP BY` operations in (Q, S) are indexed to expedite lookups and sorting. High-selectivity columns, i.e. low cardinality columns, where queries retrieve only a small subset of rows, are prioritized to minimize scan overhead.

Primary and Foreign Key Identification: Primary keys are determined based on uniqueness and non-null constraints, ensuring each row's distinct identification. Foreign keys are inferred from inter-table dependencies.

3.1.3 Foreign Key Mapping

For each column in T_i, candidate foreign key relationships are generated by forming pairs with columns from previously onboarded tables $\{T_1, T_2, \ldots, T_{i-1}\}$. An LLM evaluates these pairs based on schema similarity, including column names, datatypes, and the top frequent values, to infer potential foreign key mappings. The inferred mappings undergo manual verification, refining constraints that define permissible joins and ensuring schema consistency. To ensure computational efficiency, we impose a strict constraint that only equi-joins are considered. Given the computational complexity of joining large tables, we introduce a cost model to quantify the estimated overhead of joining T_1 and T_2. The cost function is formulated as follows:

$$C(T_1, T_2) = k_0 + k_1\big(|T_1|\log(|T_1|) + |T_2|\log(|T_2|)\big) \quad (1)$$

where k_0 and k_1 are empirically determined coefficients. This function accounts for the sorting and hashing overhead incurred during join processing. By incorporating this estimated cost, we can systematically prioritize efficient join paths, thereby mitigating excessive computational overhead associated with large table joins.

3.1.4 Caching frequency column values

Certain VARCHAR, non-binary columns with low cardinality ($< 100K$ unique values) require normalization, spell correction, and formatting to ensure accurate query execution. For instance, a user querying "headphones" would fail if the table stores it as "1300 Headphone." To address this, we identify **searchable text columns** based on cardinality and datatype. For each, we extract the top X most frequent values or those in the 99.9^{th} percentile of a priority metric (e.g., sales, clicks). These are cached, and during inference, filters are matched using a modified Levenshtein distance for robust query resolution.

3.2 Inference: Generation of SQL query

We present a multi-agent LLM-based system for translating natural language queries into SQL, consisting of a schema parser, generator and deterministic error correctors.

3.2.1 Schema Parser and Filtering

We implement a schema pruning mechanism that selectively identifies the most relevant columns from the database schema to enhance query efficiency and reduce LLM context consumption. Given a natural language query q and database $D = \{T_i = \{c_{ij}, 1 \leq j \leq |T_i|\}, 1 \leq i \leq |D|\}$, where c_{ij} represents the j^{th} column of the i^{th} table in D, we employ a ranking-enhanced encoder adapted from RESDSQL to compute relevance scores. The encoder with a softmax layer processes query q against each schema element and outputs a relevance score r_{ij} for each column c_{ij}. Columns with scores exceeding a predefined threshold δ ($r_{ij} > \delta$) are retained in the filtered schema. This pruned schema is then incorporated into the LLM's prompt context, significantly reducing input token consumption while preserving essential schema information.

3.2.2 SQL Generator

In industrial settings, SQL queries required by analysts often adhere to template-based patterns, typically requiring minor modifications such as adding filters, merging existing queries, or adjusting parameter values. Our analysis of 14,000 SQL queries revealed that merely 350 unique SQL templates accounted for 13.1k queries (>93.5% coverage). This observation underpins our hypothesis that for the vast majority of cases (>90%), SQL generation can be reliably accomplished by leveraging matching templates from historical data. For the remaining novel cases, we employ a more sophisticated methodology involving user intent comprehension, information retrieval via RAG agents, and SQL generation through a dynamic, iterative approach.

Match and Generate: We maintain a repository of verified examples and formulas, referred to as the *Example Bank*, whose creation and upkeep are discussed in Section 3.3. The Example Bank is denoted as $E = \{(q_i, s_i) \mid 1 \leq i \leq n_e\}$, where each pair (q_i, s_i) consists of a user query or keyword and its corresponding verified SQL or formula. Let e_i represent the text embedding of the noun-masked q_i, and e represent the embedding of the noun-masked user query q. The process of noun masking (detailed in Appendix A.1) replaces key nouns in the query to enhance retrieval precision. The nearest k examples from the Example Bank are selected based on cosine similarity (between $\{e_i\}$ and e), provided their similarity score exceeds a predefined threshold T. These examples are included in the LLM prompt as few-shot exemplars. If suitable examples are found, the SQL Generator is tasked with generating the SQL s. The prompt of this LLM comprises a task description outlining the objective, general guidelines, the table schemas derived during table onboarding, specific instructions such as handling date computations, formula applications, and other domain-specific rules, along with the nearest k-shot examples.

Think and Generate: If no example surpasses the threshold T, the system falls back to a more computationally intensive three-phase SQL generation process. If an agent enters this phase during inference, we only allow a deeper search.

Phase 1 (Planning/Debugging Agent): The Planning LLM performs two key tasks simultaneously. First, it generates a set of clarification questions required to generate the SQL, such as business-related formulas, concepts, abbreviations, etc. Second, it decomposes the user query into multiple ordered subtasks. An Answering agent, which has access to internal documents or the web,

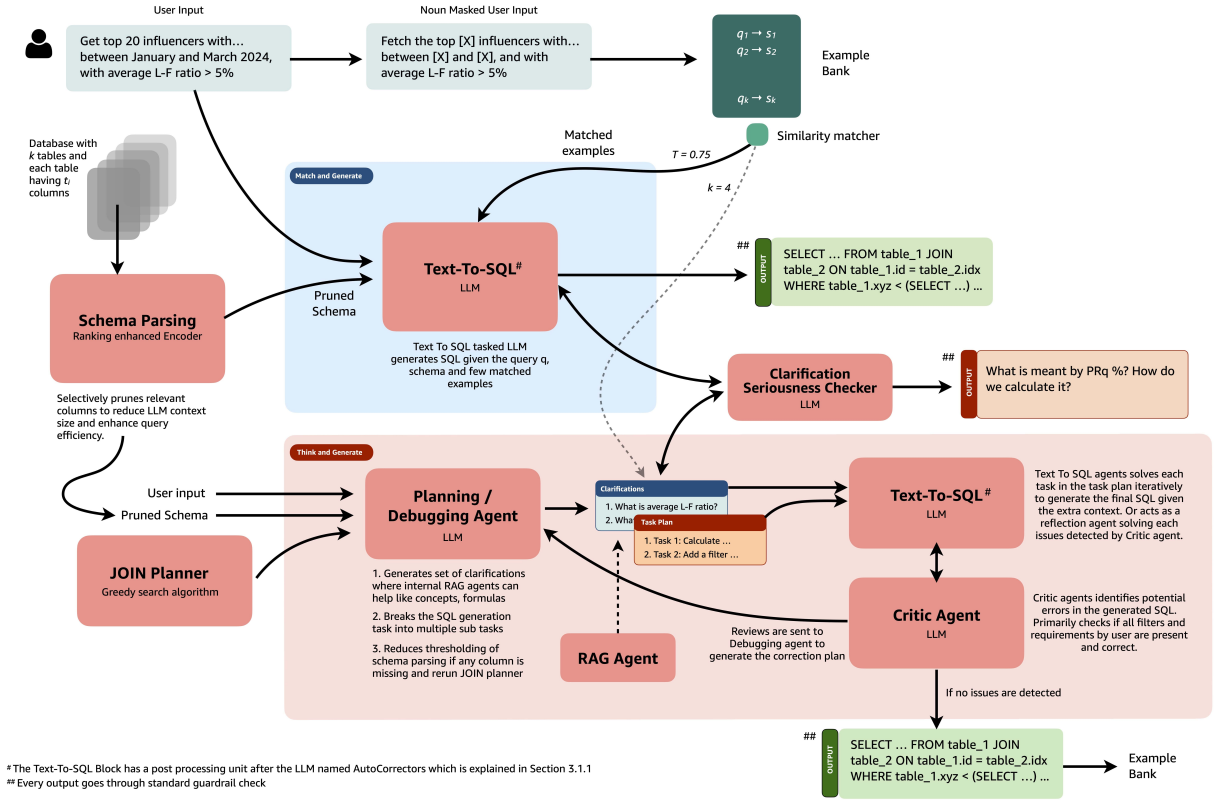

Figure 1: The end to end proposed pipeline of SQL-Genie

is leveraged to answer each clarification question. This agent also acts as a debugging planner when provided with a set of critics.

Phase 2 (Generation Agent): Using the answers to all clarification questions and the task plan, it builds an SQL query for each task. The JOIN Planner acts on each SQL query to generate and provide alternate JOIN plans. Finally, an LLM is instructed to construct an efficient SQL query.

Phase 3 (Critic Agent): A critic agent evaluates the generated SQL against the input query q. It identifies a list of potential errors and provides an explanation/review for each issue.

These errors are then fed back to the Debugging Agent for a refinement plan. For each identified issue, the Planning Agent either provides a clarification response to the Critic Agent and ignores the review or creates a requirement list of external data and a new task plan. Subsequent agents act on these, and the SQL query is updated. The Critic Agent then reevaluates it. This iterative process continues up to a maximum of m interactions. After reaching this limit, the SQL query with the fewest errors is selected as the final output.

3.2.3 AutoCorrectors

Our system employs multiple correction mechanisms to enhance query robustness. For searchable text columns, we maintain a repository of frequent values and replace filter values with close matches using modified Levenshtein distance metrics, transforming queries like `name = "rockpot"` into `name IN ("Rockpot LLC", "Rockpotロックポット", ...)`. This transformation is crucial, as the SQL Generator lacks direct access to the column's full value space, often leading to mismatches that would otherwise yield empty query results. For mathematical and date computations, the SQL Generator produces Python expressions (e.g., `<python>(datetime.now()-relativedelta(weeks=6)).strftime('%Y-%m-%d')</python>`) that are evaluated at runtime. We enforce system-wide constraints through defaults, including a 500-row limit and mandatory columns in the `SELECT` clause. To prevent type mismatch errors, our datatype matching mechanism automatically casts values to match schema definitions. Finally, a lightweight validation process executes queries on small dummy tables to catch and correct syntax errors. These autocorrection layers significantly improve query success rates in production environments. Further details on these mechanisms are provided in Appendix A.2.

3.3 Feedback Augmentation

Every novel SQL query that has been validated by the Critic agent—meaning all identified errors have been resolved—is stored in the Example Bank for future use. For queries that receive negative feedback from the user or result in an early-stopped SQL with unresolved errors, the system takes additional steps after responding to the user. Specifically, the three-phase SQL generation process is rerun multiple times with a higher temperature and deeper CoT process. The results are then ensembled to produce a more accurate SQL query in the background, ensuring that an improved version is available for future queries. Each verified SQL examples are also parsed by an LLM to generate tuple of metric, formula and the column dependency.

4 Experiments

4.1 Dataset

Internal: Our dataset consists of 18 tables, where edges indicate valid primary key-foreign key (PK-FK) relationships, and vertex size reflects the number of columns per table. We curated two datasets belonging to these tables: **Dataset-1**: Contains approximately 2,460 questions paired with manually verified SQL queries from a realistic setup. **Augmented Dataset-2**: Comprises around 14k SQL queries from various use cases, fed to LLM to generate natural language (NL) queries. Manual verification of a 200-sample subset yielded $\approx 98.5\%$ accuracy.

External: For robustness evaluation, we tested our model on three public benchmarks: WikiSQL (Zhong et al., 2017b), Spider-Test (Yu et al., 2019b) and BIRD (Li et al., 2024).

4.2 Evaluation metrics

Execution result metrics evaluate the correctness of a SQL query by comparing its execution results on the target database with the expected results.

Execution Accuracy (EX) gauges the accuracy of a predicted SQL query by executing it and comparing the results with the ground truth.

Valid Efficiency Score (VES) [Appendix A.3] measures the efficiency of valid SQL queries whose results exactly match the ground truth. We average VES over 10 runs per example.

4.3 Benchmarking

Models: We comprehensively evaluate our proposed pipeline against two categories of competitive baselines.

1. **Single-shot LLM models:** We benchmark against state-of-the-art large language models that generate SQL in a single inference pass, including GPT-4o (OpenAI, 2024), Claude 3.5 Haiku (Anthropic, 2024a), Claude 3.5 Sonnet (Anthropic, 2024b), Claude 3.7 (Anthropic, 2025), DeepSeek Coder (Guo et al., 2024), and SQLCoder-70B (Srivastava et al., 2024).

2. **Multi-LLM pipeline approaches:** We compare against recent methods that decompose text-to-SQL generation into sequential subtasks. Specifically, we benchmark RESDSQL (Li et al., 2023), which separates schema linking and SQL parsing via ranking-enhanced encoding and skeleton-aware decoding; DAIL-SQL (Gao et al., 2023), which employs iterative decomposition with verification; CHESS (Talaei et al., 2024), a multi-agent framework for retrieval, schema selection, query generation, and validation; and MAC-SQL (Wang et al., 2025), which leverages a decomposer agent for few-shot reasoning and auxiliary agents for query refinement.

Ablation Study: To analyze the contribution of individual components within our pipeline, we conduct a systematic ablation study by selectively removing each component while keeping the rest of the architecture intact. Specifically, we examine: (1) the impact of our schema pruning mechanism by replacing it with full schema passing; (2) the effect of changing T and not going through novel SQL generation route of Planning/Generation/Critic Agent and limiting the system to a single generation attempt. This ablation methodology allows us to quantify the incremental performance gains attributed to each component across our evaluation datasets.

5 Results and Discussion

As demonstrated in Tables 1 and 2, our SQLGenie framework consistently outperforms both zero-shot LLM approaches and existing multi-LLM pipelines across all evaluated datasets. On our internal production dataset, SQLGenie (Think) achieves 84.6% execution accuracy, representing a significant improvement of 21.5% over the best single-LLM baseline (Claude 3.7) and 5.3% over the strongest pipeline competitor (RESDQL). Similarly, on external benchmarks, SQLGenie estab-

Model	Dataset-1			Dataset-2		
	EX (%)	VES (%)	T_{gen} (s)	EX (%)	VES (%)	T_{gen} (s)
Zero-shot LLM models						
DeepSeekCoder	50.3	88.7	5.61	76.2	95.5	3.99
SQLCoder-70B	49.9	89.5	8.05	79.4	95.3	5.32
GPT-4o	58.4	87.0	7.56	82.1	94.8	5.65
Claude 3.5 Haiku	54.5	87.8	6.04	74.3	94.0	5.26
Claude 3.5 Sonnet	58.2	86.6	9.82	80.8	95.3	7.01
Claude 3.7	63.1	88.2	14.4	83.5	95.1	12.4
Multi-LLM pipeline approaches						
MAC-SQL	77.2	89.2	30.6	92.4	95.0	32.4
CHESS	76.6	88.4	27.4	91.2	94.5	36.8
DAIL-SQL	78.7	88.5	40.8	92.7	94.7	40.1
RESDQL	79.3	90.0	27.1	93.0	94.2	28.5
SQLGenie (Hybrid)	81.5	**93.6**	13.9	93.3	97.0	10.4
SQLGenie (Think)	**84.6**	93.6	48.7	**94.6**	**98.7**	34.7

Table 1: Performance evaluation of text-to-SQL models on internal datasets. The table compares execution accuracy (EX), valid efficiency score (VES), and generation time (T_{gen}) across zero-shot LLMs and multi-LLM pipeline approaches on both Dataset-1 and Dataset-2. SQLGenie variants demonstrate superior performance, with the Think variant achieving the highest accuracy (84.6% on Dataset-1, 94.6% on Dataset-2) while the Hybrid variant maintains competitive generation times.

Model	WikiSQL		Spider-Test		BIRD	
	EX (%)	T_{gen} (s)	EX (%)	T_{gen} (s)	EX (%)	T_{gen} (s)
Zero-shot LLM models						
DeepSeekCoder	78.4	4.19	66.6	5.56	49.8	6.04
SQLCoder-70B	70.2	5.03	65.4	7.27	47.2	9.87
GPT-4o	81.5	6.18	71.5	8.05	53.5	8.96
Claude 3.5 Haiku	75.1	5.84	64.8	6.41	52.1	7.13
Claude 3.5 Sonnet	83.5	7.20	70.4	9.18	55.6	10.7
Claude 3.7	86.9	12.2	76.7	13.6	61.3	14.4
Multi-LLM pipeline approaches						
MAC-SQL	87.9	22.7	81.2	36.4	63.7	38.1
CHESS	88.1	20.4	82.7	33.7	67.4	30.5
DAIL-SQL	92.0	48.8	**84.3**	44.6	67.8	62.6
RESDQL	91.4	14.7	78.4	30.3	70.1	28.7
SQLGenie (Think)	**92.8**	15.3	82.1	40.6	**73.8**	50.8

Table 2: Performance comparison on external benchmark datasets. We report Execution Accuracy (EX) and SQL generation time (T_{gen}). SQLGenie demonstrates robust generalization capabilities, achieving state-of-the-art performance on BIRD, Spider and WikiSQL.

lishes new state-of-the-art performance with 92.8% accuracy on WikiSQL and 73.8% on the more challenging BIRD dataset. Notably, our hybrid variant strikes an optimal balance between accuracy and efficiency, achieving competitive execution accuracy (81.5% on production data) while maintaining generation times comparable to single-LLM approaches (T_{gen} = 14.6s). The performance differential is particularly pronounced on complex queries involving multiple tables and nested operations, where our schema pruning mechanism and multi-agent collaboration demonstrate their efficacy. Analysis of the Valid Efficiency Score (VES) further reveals that the use of JOIN planner in SQLGenie not only helps it generates more accurate queries but also produces more efficient SQL, with a 3.6% improvement over the best baseline on our production dataset.

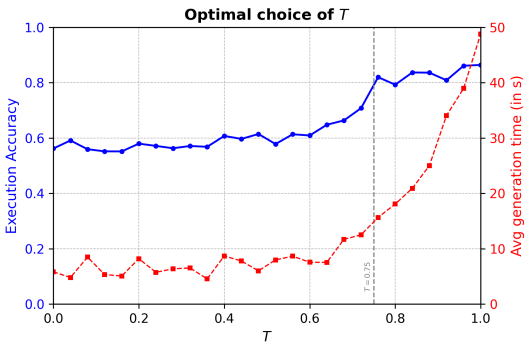

Figure 2: Impact of T on execution accuracy (blue) and generation time (red). $T = 0$ represents unconstrained example selection, while $T = 1$ enforces structured reasoning. Higher T improves accuracy but increases latency. The dashed line at $T = 0.75$ marks a trade-off point, chosen based on Dataset-1.

Our ablation studies reveal several key insights into SQLGenie's performance advantages. Replacing our schema pruning component with full schema passing decreases execution accuracy by $\approx 4.6\%$ on the internal dataset and $\approx 3.5\%$ on the external dataset, while increasing input token length by $\approx 65\%$, which in turn raises generation time by $\approx 41\%$. As shown in Figure 2, execution accuracy remains relatively stable across different values of T, but generation time rises sharply beyond $T = 0.75$. This suggests that setting T too high can significantly impact latency without substantial accuracy gains. More results are presented in the Appendix.

6 Conclusion

In this paper, we presented SQLGenie, a practical system for reliable SQL generation that addresses the challenges of ambiguous user intent and database constraints. Our comprehensive approach integrates intelligent table onboarding, multi-agent SQL generation, and feedback augmentation to achieve state-of-the-art performance. Experimental results demonstrate that SQLGenie outperforms existing methods on both internal and external benchmarks, while reducing generation time by 64%. Future work will focus on extending SQLGenie to handle more complex analytical queries involving window functions and recursive CTEs, as well as exploring cross-database query generation to support federated analytics scenarios.

References

Anthropic. 2024a. Claude 3.5 haiku. https://www.anthropic.com/claude/haiku. Released October 22, 2024. Available at https://www.anthropic.com/claude/haiku.

Anthropic. 2024b. Claude 3.5 sonnet (upgraded). https://www.anthropic.com/news/claude-3-5-sonnet. Upgraded version released October 22, 2024. Available at https://www.anthropic.com/news/claude-3-5-sonnet.

Anthropic. 2025. Claude 3.7 sonnet. https://www.anthropic.com/news/claude-3-7-sonnet. Released February 24, 2025. Available at https://www.anthropic.com/news/claude-3-7-sonnet.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2016. Neural machine translation by jointly learning to align and translate. *Preprint*, arXiv:1409.0473.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Preprint*, arXiv:1810.04805.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *Preprint*, arXiv:1905.08205.

Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a natural language interface to complex data. *ACM Trans. Database Syst.*, 3(2):105–147.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. *Preprint*, arXiv:2302.05965.

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *Preprint*, arXiv:1907.11692.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Chen Shen, Jin Wang, Sajjadur Rahman, and Eser Kandogan. 2024. Demonstration of a multi-agent framework for text to sql applications with large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 5280–5283, New York, NY, USA. Association for Computing Machinery.

Rishabh Srivastava, Wendy Aw, and Wong Jing Ping. 2024. Sqlcoder-70b. https://huggingface.co/defog/sqlcoder-70b-alpha. Accessed March 22, 2025.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

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. 2019a. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

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. 2019b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017a. Seq2sql: Generating structured queries from natural language using reinforcement learning. *Preprint*, arXiv:1709.00103.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017b. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Appendix

A.1 Noun-Masking

To enhance retrieval precision when matching examples with user input, we implement a noun-masking mechanism utilizing a T5-based model fine-tuned on named entity recognition (NER) datasets. This approach identifies schema-independent nouns, proper nouns, numerical values, and entity codes in natural language queries, replacing them with a standardized `[MASK]` token. The resulting abstracted query functions as a generic template, effectively capturing the structural intent while eliminating entity-specific variations. For instance, semantically equivalent queries like `"Get reel counts of top influencers aged **18** residing in **San Francisco**"` and `"Get reel counts of top influencers aged **30** residing in **Tokyo**"` are both transformed into the template `"Get reel counts of top influencers aged **[MASK]** residing in **[MASK]**"`, achieving 100% similarity in our embedding space. This normalization significantly improves the robustness of our retrieval system by focusing on query structure rather than specific entity values, thereby facilitating more accurate template matching and subsequent SQL generation.

A.2 Auto-Correctors

Search: We maintain a repository of frequently occurring values for all searchable text columns. When filter values appear in the `WHERE` clause, the system searches this repository to identify the closest matches using a modified Levenshtein distance metric (`rapidfuzz.fuzz.WRatio`). If a near match is found, the filter value is replaced accordingly. For instance, `name = "rockpot"` is rewritten as `name IN ("Rockpot LLC", "Rockpot", "Rockpott (ロックポット)")`, while `music_genre NOT IN ("calm", "sleepy")` is transformed into `music_genre NOT IN ("Calm 1860", "Calm 2025 [Updated]", "Sleepy time", "Sleep")`. This transformation is crucial, as the SQL Generator lacks direct access to the column's full value space, often leading to mismatches that would otherwise yield empty query results.

Date/Math Computation: The SQL Generator, whether operating with or without examples, is prompted to generate Python expressions in cases involving numerical calculations or date computations. These expressions follow the format `<python>89.8*16/100</python>` or `<python>(datetime.now()-relativedelta(weeks=6)).strftime('%Y-%m-%d')</python>`. A parser evaluates the generated expression using Python's `eval` function and replaces the placeholder with the computed result.

Defaults: To enforce system-wide constraints, a default `LIMIT` of 500 is applied when the user does not specify a count. Additionally, a predefined set of mandatory columns is appended to the `SELECT` clause if no `GROUP BY` operation is present. When a `GROUP BY` clause exists, these mandatory columns are included using their respective aggregation functions.

Datatype Matching: The LLM sometimes hallucinates and assumes fields like student roll numbers or country indices are integers based on general knowledge, overlooking schema definitions. For instance, even if `student_roll_number` is a `VARCHAR`, it may generate an invalid filter like `student_roll_number = 4`. A deterministic type-correction mechanism prevents such errors by casting values appropriately, e.g., rewriting it as `student_roll_number = "4"` based on the schema.

Dummy Testing: We apply a lightweight validation mechanism to ensure query syntax correctness by executing the SQL on small dummy tables (<10 rows). If syntax errors occur, an LLM-based correction agent automatically rectifies them. In practice, such failures are rare, but this safeguard ensures robustness in edge cases.

A.3 Evaluation metrics

A.3.1 Valid Efficiency Score (VES)

For a dataset with N examples, VES is computed as:

$$VES = \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}(V_n, \hat{V}_n) \cdot R(Y_n, \hat{Y}_n), \quad (2)$$

where $\hat{Y}_n$ and $\hat{V}_n$ are the predicted query and results, and Y_n and V_n are the ground truth. The indicator function is:

$$\mathbb{1}(V_n, \hat{V}_n) = \begin{cases} 1, & V_n = \hat{V}_n \\ 0, & V_n \neq \hat{V}_n \end{cases} \quad (3)$$

Then,

$$R(Y_n, \hat{Y}_n) = \sqrt{\frac{E(Y_n)}{E(\hat{Y}_n)}} \tag{4}$$

represents the relative execution efficiency, where $E(\cdot)$ is the execution time.