AGENTIQL: An Agent-Inspired Multi-Expert Framework for Text-to-SQL Generation

Omid Reza Heidari*
Concordia University
Montreal, Canada
o_heidar@live.concordia.ca

Siobhan Reid
Concordia University
Montreal, Canada
si_reid@live.concordia.ca

Yassine Yaakoubi
Concordia University
Montreal, Canada
yassine.yaakoubi@concordia.ca

Abstract

LLMs have advanced text-to-SQL generation, yet monolithic architectures struggle with complex reasoning and schema diversity. We propose AGENTIQL, an agent-inspired multi-expert framework that combines a reasoning agent for question decomposition, a coding agent for sub-query generation, and a refinement step for column selection. An adaptive router further balances efficiency and accuracy by selecting between our modular pipeline and a baseline parser. Several steps in the pipeline can be executed in parallel, making the framework scalable to larger workloads. Evaluated on the Spider benchmark, AGENTIQL improves execution accuracy and interpretability and achieves up to 86.07% EX with 14B models using the Planner&Executor merging strategy. The attained performance is contingent upon the efficacy of the routing mechanism, thereby narrowing the gap to GPT-4-based SOTA (89.65% EX) while using much smaller open-source LLMs. Beyond accuracy, AGENTIQL enhances transparency by exposing intermediate reasoning steps, offering a robust, scalable, and interpretable approach to semantic parsing.

1 Introduction

Natural language to SQL (NL2SQL) technology, which enables the transformation of everyday language queries into structured SQL commands, marks a major step forward in improving data accessibility. It empowers both novice and expert users to efficiently extract meaningful insights from large and complex datasets [1–12]. Recent progress in large language models has considerably improved the efficacy and accuracy of NL2SQL systems, but key challenges remain. Monolithic LLM architectures often struggle with complex reasoning and with handling diverse database schemas, while static ensemble methods introduce significant computational overhead. Serving massive large language models (LLMs) in real-world environments is also costly and impractical for many applications. Moreover, most existing systems offer limited interpretability, making it difficult to understand *why* a particular SQL query was generated or to identify misalignment.

To address these limitations, we propose AGENTIQL, an agent-inspired, multi-expert architecture for NL2SQL. Instead of relying on a single monolithic LLM, AGENTIQL decomposes query generation into specialized expert components and employs a learned router to balance accuracy and efficiency. This design yields a more interpretable, modular, and scalable framework for structured query

*Website: https://omid-reza.github.io, alternative email: omid.orh@gmail.com

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Efficient Reasoning.

generation. Our contributions are threefold: (i) a **Divide-and-Merge** module, where a reasoning agent decomposes natural language questions into sub-questions and a coding agent generates corresponding sub-queries that are then merged, improving interpretability through visible intermediate steps; (ii) a **Column Selection** (CS) refinement that adjusts column choices and ordering in the final SQL query, increasing alignment with user intent and boosting execution accuracy; and (iii) an **Adaptive Routing** mechanism that directs queries (e.g. XGBoost classifier[13] or a reasoning-agent "judge"), enhancing efficiency, efficacy, and robustness by adaptively allocating available resources to query complexity.

2 Related Works

There are three main categories of LLM-based NL2SQL approaches: **prompt engineering** [14–17], supervised fine-tuning (SFT) [18], and reinforcement learning (RL)-based optimization [19].

Prompt engineering has shown strong potential for NL2SQL, particularly through zero-shot approaches (e.g., using ChatGPT/GPT-4 directly [14]) and few-shot chain-of-thought prompting techniques [20, 15, 17]. These methods leverage pretrained LLMs with carefully crafted prompts. Nonetheless, many prompt-based methods depend on multi-path generation combined with self-consistency (majority voting) to select the optimal output, which leads to considerable inference overhead in practice.

In contrast, **SFT-based** approaches fine-tune smaller or domain-specific models on NL2SQL data to generate more controllable SQL queries (e.g., CODES fine-tunes open-source LLMs for text-to-SQL [18]). However, reduced parameter capacity can limit their ability to handle complex NL2SQL reasoning or generalize effectively to databases in new domains.

3 Method

Given a labeled dataset $\mathcal{D} = \{(x_i, s_i, y_i)\}_{i=1}^{N}$, where each natural language query x_i with database schema s_i is paired with the executable SQL query y_i, the goal is to train a model that maps $(x_i, s_i) \mapsto y_i$. We introduce AGENTIQL, a multi-expert architecture designed to address this task through query decomposition, specialized code generation, and adaptive routing.

3.1 Division

Table Selection. For a given question x with database schema s, we first filter out irrelevant tables from s using a reasoning LLM f_{reason}. Formally, this produces a reduced schema $\tilde{s} = f_{\text{reason}}(x, s)$, where $\tilde{s} \subseteq s$ contains only the tables likely needed to answer x, and the semantic content of the original schema remains unchanged. The reduced schema $\tilde{s}$ is passed as context to all later stages in the pipeline.

Question Decomposition. Next, we employ another reasoning LLM f_{decomp} to decompose the natural language query x (with respect to schema $\tilde{s}$) into a set of smaller sub-questions or tasks. Formally, this step produces $\{x_1, x_2, \ldots, x_k\} = f_{\text{decomp}}(x, \tilde{s})$, where each x_j is a natural language sub-question. These sub-questions are intended such that solving each one individually (on the given schema) and then merging the results will answer the original SQL query. Decomposition breaks (complex, resource-intensive) queries into one or multiple manageable natural language sub-questions and makes the reasoning process explicit.

Query Generation: For each sub-question x_j, we use a coding LLM f_{gen} (one specialized in code/SQL generation) to produce the corresponding SQL query y_j. This model operates in a few-shot setting, similar to the baseline. Formally, the output of this step is $f_{\text{gen}}(\{x_1, x_2, \ldots, x_k\}) = \{y_1, y_2, \ldots, y_k\}$, where each y_i is the SQL query corresponding to the sub-question x_i. If an error is detected in a generated query, a refinement process is triggered. In this process, the same LLM is used to correct the erroneous query for up to R iterations. At refinement step $r \in \{1, \ldots, R\}$, the model produces $y_i^{(r)} = f_{\text{gen}}(x_i, y_i^{(r-1)})$, correcting potential errors in the previous version.

3.2 Merge

The essential part of the pipeline is the ability to merge the generated sub-queries $\{y_1, y_2, \ldots, y_k\}$ into a single final SQL query y. Formally, this is achieved through a merge function g such that $y = g(y_1, y_2, \ldots, y_k)$. We explore two strategies for g in this work: **(1) Selecting the Last Sub-query:** The output SQL query is taken to be the translation of the final generated sub-question. This strategy assumes that the reasoning agent orders sub-questions such that the last one corresponds to the complete solution. It is simple and fast, but may fail if the final sub-question does not cover the entire query. **(2) Planner&Executor:** A reasoning LLM is employed as a planner to determine how the generated sub-queries $\{y_1, \ldots, y_k\}$ should be combined. The planner produces a natural-language description or pseudocode of a merging plan, which is then executed by a coding LLM (the executor) to yield the final SQL query y. This approach is more general, as it does not assume that one sub-query fully answers the question, but it introduces additional computational overhead.

This divide-and-merge design naturally follows a *prompt chaining workflow*, in which intermediate reasoning steps are explicit and sequential. Such chaining improves interpretability by exposing sub-questions and their corresponding sub-queries, reflecting the workflow taxonomy introduced by Anthropic for building effective agents [21].

3.3 Column Selection

After merging, we obtain an intermediate SQL query $\hat{y}$. In a final refinement step, a reasoning LLM f_{col} is given the original question x, schema $\tilde{s}$, and query $\hat{y}$ from the previous step. The model then performs *CS*, adjusting the `SELECT` clause, ensuring output columns and their ordering precisely match the requirements of x. Formally, the final SQL query is obtained as $y = f_{\text{col}}(x, s, \hat{y})$ where f_{col} denotes the CS function, aligning columns in $\hat{y}$ with user intent.

3.4 Routing

After evaluating the divide-and-merge module across multiple LLMs, we identified complementary strengths in the different approaches. According to Tables 2–3, decomposition often captured domain-specific reasoning more effectively, whereas the baseline produced more consistent parsing in other cases. To exploit these strengths, we introduce a router that selects between the baseline or divide-and-merge pipeline based on the schema and query received. In addition to this intuition, we also tested a simple schema-level metric (the table count) as a proxy for complexity. As observed in Table 4, there are meaningful correlations between schema size and relative performance, suggesting that even lightweight signals can guide effective routing decisions.

The performance comparisons reported in Tables 1–3 further confirm that, although the baselines perform well on simple queries, the divide-and-merge strategies (particularly Planner and Executor with CS) are more robust when it comes to complex reasoning tasks. These findings suggest that routing based on simple complexity measures is promising, and motivate the development of more advanced routers that combine such metrics with learned decision functions to improve efficiency and accuracy in practice.

4 Experimental Results

4.1 Dataset

Among available text-to-SQL benchmarks such as BIRD [6], SQL-Eval [22], and SKYRL-SQL [23], the Spider [24] dataset is selected for our experiments. Spider was the first large-scale benchmark proposed for the text-to-SQL task and has since been widely adopted in the literature. It contains over 10,000 natural language questions across 200+ databases with diverse schemas, covering multiple domains and complex query structures. Its widespread use ensures comparability with prior work and provides a reliable basis for assessing the effectiveness of the proposed approach.

4.2 Baseline

A coding LLM serves as a standard baseline for the text-to-SQL task. Specifically, we use the Qwen2.5-Coder series as the base text-to-SQL model. The model is prompted with several question-

SQL examples and then directly generates the SQL for a new question in one step, using few-shot prompting, without any task-specific adaptation or additional training. We adopt a vanilla LLM as our baseline parser, which simply maps text to SQL based on the provided in-context examples.

4.3 Metrics and Parameters

Execution accuracy (EX) is adopted as the primary evaluation metric. A predicted SQL query is considered correct if it executes to the same result as the ground-truth query on the evaluation database. This metric, which directly measures end-task success, is standard in Spider and other NL2SQL evaluations. The SQL refinement loop for query generation is limited to $R = 3$ attempts, balancing accuracy gains with inference cost.

4.4 Qualitative Results

We first present three success cases and two failure cases. Figures 3–5 show instances where the baseline fails but our method succeeds, handling challenging scenarios such as multi-join queries, nested aggregations, and schema alignment. In contrast, Figures 6–7 illustrate two failure cases of our method: errors in sub-question decomposition or the final CS lead to incorrect outputs.

4.5 Quantitative Results

Impact of Column Selection and Merging Strategy. Table 1 evaluates the effect of the CS refinement and the two merging strategies. Incorporating the CS step consistently improves performance across model sizes and strategies, typically by 2-5% EX. For example, with 7B models using the Last Sub-query merge, adding the CS step raises accuracy from 72.26% to 74.44%. The Planner&Executor strategy benefits especially from CS: without CS it sometimes underperforms the simpler (Last Sub-query) strategy (e.g., 66.77% vs 72.26% at 7B), but with CS it surpasses it (75.85% vs 74.44% at 7B). This can be attributed to the planner occasionally introducing extraneous or misordered columns that the final column-selection step fixes. While Planner&Executor with CS yields the highest accuracy for each model size, it also incurs more latency than the Last Sub-query heuristic, highlighting a trade-off between accuracy and efficiency.

Divide-and-Merge Module and AGENTIQL. Table 3 compares our divide-and-merge module (with CS) against the baseline across different model scales. With the integration of an effective routing mechanism, performance approaches that of the state of the art (SOTA), even when smaller reasoning and coding models compared to GPT-4o are employed. We also found that the relative performance of AGENTIQL vs. the baseline correlates with database complexity: for larger LLMs, our pipeline tends to have a greater advantage on queries from schemas with many tables, whereas with a small model, the baseline performed better on the most complex schemas.

5 Conclusion

AGENTIQL demonstrates that dividing complex natural language queries into sub-questions and merging their answers can improve interpretability while maintaining high accuracy in text-to-SQL generation. The CS refinement consistently yields additional gains in execution accuracy, with the Planner&Executor merging strategy performing best when refinement is applied. Using an adaptive router to combine our pipeline with a strong baseline further enhances robustness and overall performance by exploiting their complementary strengths. Qualitative analyses show strong improvements in handling joins, nested aggregations, and schema alignment, though errors in decomposition and column refinement remain as failure cases. To facilitate reproducibility, we will release our code and prompt templates publicly.

Several limitations and directions remain for future work. We evaluated primarily on the Spider dataset; testing AGENTIQL on additional benchmarks (e.g., BIRD or SQL-Eval) will be important to assess the generalizability of the findings. Experiments were limited to open-source LLMs up to 32B parameters, while larger models (and closed-source systems such as GPT-4 or Claude) were not fully explored due to resource constraints. Scaling to very large models incurs substantial cost: for example, in a preliminary trial, using a 235B-parameter model in our pipeline took nearly 60 minutes per question on four A100 GPUs (with CPU offloading). There are multiple avenues for

future research to address the above limitations. For instance, RL could be incorporated into the query generation stage, similar to what has been demonstrated in SkyRL-SQL [19], to provide adaptive feedback and further enhance the quality of generated SQL queries. Testing should also be extended to additional datasets such as BIRD and SQL-Eval, in order to evaluate the robustness of the approach across different benchmarks. Furthermore, experiments with closed-source LLMs, such as GPT-4o [25] and Claude 4 [26], as well as larger open-source models, such as Qwen3-235B-A22B-Instruct and Qwen3-Coder-480B-A35B-Instruct, will provide deeper insights into the framework's scalability and general applicability. The effect of different merging strategies on accuracy and efficiency should be investigated, as should various routing options, such as XGBoost classifier and retrieval-augmented generation [27], to enable the framework to adapt more effectively to diverse question complexities and schema structures.

References

[1] Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. Text-to-sql error correction with language models of code, 2023. URL https://arxiv.org/abs/2305.13073.

[2] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of text-to-sql in the era of llms: Where are we, and where are we going?, 2025. URL https://arxiv.org/abs/2408.05109.

[3] Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. Exploring chain-of-thought style prompting for text-to-sql, 2023. URL https://arxiv.org/abs/2305.14215.

[4] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers, 2021. URL https://arxiv.org/abs/1911.04942.

[5] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. Few-shot text-to-sql translation using structure and content prompt learning. *Proc. ACM Manag. Data*, 1 (2), June 2023. doi: 10.1145/3589292. URL https://doi.org/10.1145/3589292.

[6] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2023. URL https://arxiv.org/abs/2305.03111.

[7] Aiwei Liu, Xuming Hu, Li Lin, and Lijie Wen. Semantic enhanced text-to-sql parsing via iteratively learning schema linking graph, 2022. URL https://arxiv.org/abs/2208.03903.

[8] Lihan Wang, Bowen Qin, Binyuan Hui, Bowen Li, Min Yang, Bailin Wang, Binhua Li, Fei Huang, Luo Si, and Yongbin Li. Proton: Probing schema linking information from pre-trained language models for text-to-sql parsing, 2022. URL https://arxiv.org/abs/2206.14017.

[9] Mohammadreza Pourreza, Ruoxi Sun, Hailong Li, Lesly Miculicich, Tomas Pfister, and Sercan O. Arik. Sql-gen: Bridging the dialect gap for text-to-sql via synthetic data and model merging, 2024. URL https://arxiv.org/abs/2408.12733.

[10] Mohammadreza Pourreza and Davood Rafiei. Dts-sql: Decomposed text-to-sql with small large language models, 2024. URL https://arxiv.org/abs/2402.01117.

[11] Ruoxi Sun, Sercan Ö. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. Sql-palm: Improved large language model adaptation for text-to-sql (extended), 2024. URL https://arxiv.org/abs/2306.00739.

[12] Daking Rai, Bailin Wang, Yilun Zhou, and Ziyu Yao. Improving generalization in language model-based text-to-sql semantic parsing: Two simple semantic boundary-based techniques, 2023. URL https://arxiv.org/abs/2305.17378.

[13] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794. ACM, August 2016. doi: 10.1145/2939672.2939785. URL http://dx.doi.org/10.1145/2939672.2939785.

[14] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. C3: Zero-shot text-to-sql with chatgpt, 2023. URL https://arxiv.org/abs/2307.07306.

[15] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation, 2023. URL https://arxiv.org/abs/2308.15363.

[16] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql, 2024. URL https://arxiv.org/abs/2410.01943.

[17] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction, 2023. URL https://arxiv.org/abs/2304.11015.

[18] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql, 2024. URL https://arxiv.org/abs/2402.16347.

[19] Shu Liu, Sumanth Hegde, Shiyi Cao, Alan Zhu, Dacheng Li, Tyler Griggs, Eric Tang, Akshay Malik, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-sql: Matching gpt-4o and o4-mini on text2sql with multi-turn rl, 2025. Notion Blog.

[20] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. Mac-sql: A multi-agent collaborative framework for text-to-sql, 2025. URL https://arxiv.org/abs/2312.11242.

[21] Anthropic. Building effective agents. https://www.anthropic.com/engineering/building-effective-agents, 2024. Accessed: 2025-09-03.

[22] defog-ai. sql-eval: Evaluate the accuracy of LLM generated outputs. https://github.com/defog-ai/sql-eval, 2025. Accessed: 2025-09-03.

[23] NovaSky-AI. SkyRL-v0-80-data. https://huggingface.co/datasets/NovaSky-AI/SkyRL-v0-80-data, 2025. Dataset. Accessed: 2025-09-03.

[24] 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. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019. URL https://arxiv.org/abs/1809.08887.

[25] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, and et al. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

[26] Anthropic. Claude 4. https://www.anthropic.com/news/claude-4, 2024. Large language model.

[27] 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. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

[28] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql, 2025. URL https://arxiv.org/abs/2411.08599.

[29] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895. doi: 10.1098/rspl.1895.0041.

[30] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. doi: 10.2307/1412159.

A Technical Appendices and Supplementary Material

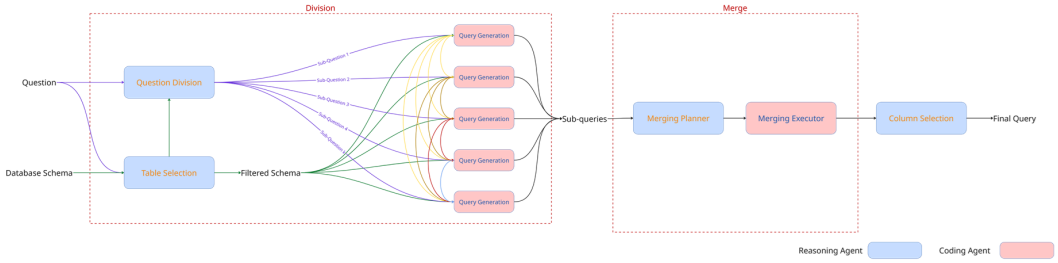

Figure 1: The Divide-and-Merge module of AGENTIQL. The reasoning agent splits an input natural language query into multiple sub-questions. The coding agent then generates a corresponding SQL sub-query for each sub-question, and finally all sub-queries are merged to produce the final SQL query. This multi-step approach explicitly exposes intermediate reasoning steps and ensures the final query aligns with the question's intent.

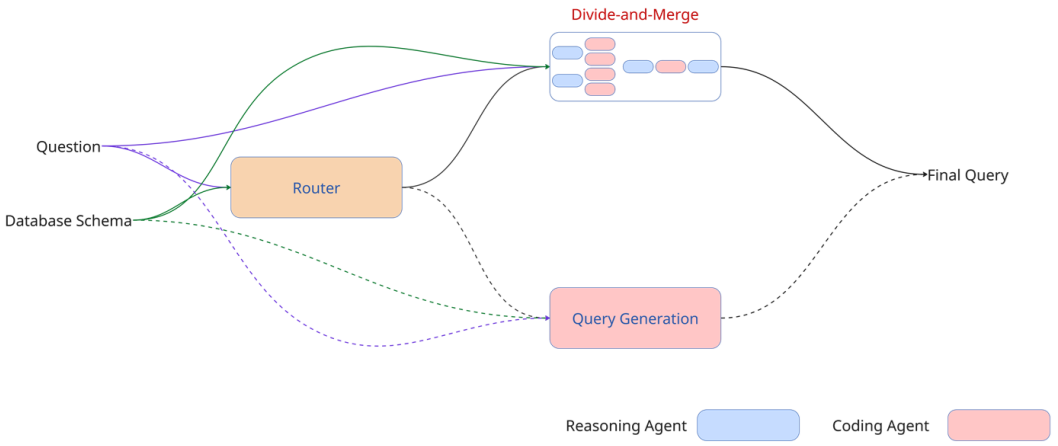

Figure 2: Overall architecture of AGENTIQL. An input query is first evaluated by an adaptive router, which decides whether to send it to a one-step baseline parser or to the divide-and-merge module. The baseline directly generates the SQL query, while the divide-and-merge module processes the query through a multi-expert pipeline before producing the final SQL. This design balances interpretability, provided by the modular pipeline, with efficiency, provided by direct execution for simpler cases.

Question: List the id, first name and last name of the customers who both have placed more than 2 orders and have bought at least 3 items.

Database Schema:

- Products (product_id, parent_product_id, product_name, product_price, product_color, product_size, product_description)
- Customers (customer_id, gender_code, customer_first_name, customer_middle_initial, customer_last_name, email_address, login_name, login_password, phone_number, address_line_1, town_city, county, country)
- Customer_Payment_Methods (customer_id, payment_method_code)
- Invoices (invoice_number, invoice_status_code, invoice_date)
- Orders (order_id, customer_id, order_status_code, date_order_placed)
- Order_Items (order_item_id, product_id, order_id, order_item_status_code)
- Shipments (shipment_id, order_id, invoice_number, shipment_tracking_number, shipment_date)
- Shipment_Items (shipment_id, order_item_id)

Sub-question 1: Find customers who have placed more than 2 orders.
Sub-question 2: Join this result with the Orders table to get order details.
Sub-question 3: Count the number of orders per customer.
Sub-question 4: Filter to customers who have placed more than 2 orders.
Sub-question 5: Find customers who have bought at least 3 items.
Sub-question 6: Join this result with the Order_Items table to get item details.
Sub-question 7: Count the number of items per customer.
Sub-question 8: Filter to customers who have bought at least 3 items.
Sub-question 9: Find the intersection of customers from steps 4 and 8.
Sub-question 10: Retrieve the id, first name, and last name of these customers.

Divide-and-Merge SQL:

```
SELECT customer_id, customer_first_name, customer_last_name FROM Customers
WHERE customer_id IN (
    SELECT c.customer_id FROM Customers c JOIN Orders o ON c.customer_id = o.
        customer_id GROUP BY c.customer_id HAVING COUNT(o.order_id) > 2
    INTERSECT
    SELECT c.customer_id FROM Customers c JOIN Orders o ON c.customer_id = o.
        customer_id JOIN Order_Items oi ON o.order_id = oi.order_id GROUP BY c.
        customer_id HAVING COUNT(oi.order_item_id) >= 3
);
```

Baseline SQL:

```
SELECT c.customer_id, c.customer_first_name, c.customer_last_name FROM Customers c
    JOIN Orders o ON c.customer_id = o.customer_id GROUP BY c.customer_id HAVING COUNT(
    o.order_id) > 2
INTERSECT
SELECT c.customer_id, c.customer_first_name, c.customer_last_name FROM Customers c
    JOIN Order_Items oi ON c.customer_id = oi.customer_id GROUP BY c.customer_id
    HAVING COUNT(oi.order_item_id) >= 3;
```

Invalid join path: incorrect linking of Customers directly to Order_Items

Figure 3: Success case with Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct using the Planner&Executor merging strategy. The query requires finding customers with both more than two orders and at least three items. The baseline SQL fails by joining `Customers` directly with `Order_Items`, violating the schema. The Divide-and-Merge approach, however, decomposes the task, routes through `Orders`, and intersects constraints, producing a valid SQL that retrieves the correct customers.

Question: What is the price of all products being ordered on average?

Database Schema:

- Products (product_id, parent_product_id, product_name, product_price, product_color, product_size, product_description)
- Customers (customer_id, gender_code, customer_first_name, customer_middle_initial, customer_last_name, email_address, login_name, login_password, phone_number, address_line_1, town_city, county, country)
- Customer_Payment_Methods (customer_id, payment_method_code)
- Invoices (invoice_number, invoice_status_code, invoice_date)
- Orders (order_id, customer_id, order_status_code, date_order_placed)
- Order_Items (order_item_id, product_id, order_id, order_item_status_code)
- Shipments (shipment_id, order_id, invoice_number, shipment_tracking_number, shipment_date)
- Shipment_Items (shipment_id, order_item_id)

Sub-question 1: Find the price of each product.
Sub-question 2: Join the Orders and Order_Items tables to associate orders with their items.
Sub-question 3: Join the resulting dataset with the Products table to get the prices of the ordered products.
Sub-question 4: Calculate the average price of the ordered products.

Divide-and-Merge SQL:

```
SELECT AVG(p.product_price) AS average_price
FROM Order_Items oi
JOIN Products p ON oi.product_id = p.product_id;
```

Baseline SQL:

```
SELECT AVG(product_price)
FROM Products;
```

Ignores orders: averages all products (missing join to Order_Items)

Figure 4: Success case with Qwen2.5-14B-Instruct and Qwen2.5-Coder-14B-Instruct using the Last-Subquery merging strategy. The query asks for the average price of products that have been ordered. The baseline SQL incorrectly averages all products in `Products`, ignoring order information. In contrast, the Divide-and-Merge approach decomposes the task, joins `Order_Items` with `Products`, and computes the average over ordered products only, yielding the correct result.

Question: Please show the most common affiliation for city channels.

Database Schema:

- city_channel (ID, City, Station_name, Owned_Since, Affiliation)
- radio (Radio_ID, Transmitter, Radio_MHz, 2FM_MHz, RnaG_MHz, Lyric_FM_MHz, ERP_kW)
- tv_show (tv_show_ID, tv_show_name, Sub_tittle, Next_show_name, Original_Airdate)
- city_channel_radio (City_channel_ID, Radio_ID, Is_online)
- city_channel_tv_show (City_channel_ID, tv_show_ID, Is_online, Is_free)

Sub-question 1: Select the Affiliation column from the city_channel table.
Sub-question 2: Select the Affiliation column from the city_channel table.
Sub-question 3: Group the results by Affiliation.
Sub-question 4: Count the number of occurrences for each Affiliation.
Sub-question 5: Order the groups by the count in descending order.
Sub-question 6: Select the top group's Affiliation.

Divide-and-Merge SQL:

```
SELECT Affiliation
FROM (
    SELECT Affiliation, COUNT(*) AS count
    FROM city_channel
    GROUP BY Affiliation
    ORDER BY count DESC
) AS grouped_affiliations
LIMIT 1;
```

Baseline SQL:

```
SELECT Affiliation, COUNT(*) AS count
FROM city_channel
GROUP BY Affiliation
ORDER BY count DESC
LIMIT 1;
```

Extra output: baseline returns counts instead of only the affiliation

Figure 5: Success case with Qwen2.5-32B-Instruct and Qwen2.5-Coder-32B-Instruct using the Planner&Executor merging strategy. The query asks for the most common affiliation among city channels. The baseline SQL outputs both the affiliation and its count, adding extra information. The Divide-and-Merge approach refines the output to return only the affiliation, exactly matching the query intent.

Question: Show agency ids and the number of clients for each agency.

Database Schema:

- Agencies (agency_id, agency_details)
- Staff (staff_id, agency_id, staff_details)
- Clients (client_id, agency_id, sic_code, client_details)
- Invoices (invoice_id, client_id, invoice_status, invoice_details)
- Meetings (meeting_id, client_id, meeting_outcome, meeting_type, billable_yn, start_date_time, end_date_time, purpose_of_meeting, other_details)
- Payments (payment_id, invoice_id, payment_details)
- Staff_in_Meetings (meeting_id, staff_id)

Sub-question 1: Join the Agencies table with the Clients table on agency_id.
Sub-question 2: Count the number of clients for each agency.
Sub-question 3: Select the agency_id and the count of clients.

Divide-and-Merge SQL:

```
SELECT agency_id, COUNT(client_id) AS client_count
FROM Agencies a
JOIN Clients c ON a.agency_id = c.agency_id
GROUP BY agency_id;
```

Ambiguous column: agency_id not qualified after join

Baseline SQL:

```
SELECT agency_id, COUNT(client_id) AS client_count
FROM Clients
GROUP BY agency_id;
```

Figure 6: Failure case with Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct using the Planner&Executor merging strategy. The query asks for agency IDs with their client counts. The Divide-and-Merge SQL introduces an explicit join between `Agencies` and `Clients`, but fails due to an ambiguous reference to `agency_id`. The baseline SQL, grouping directly over `Clients`, avoids this error and produces the correct result with a simpler query.

Question: Return the number of stories for each building in the region named “Abruzzo”.

Database Schema:

- building (Building_ID, Region_ID, Name, Address, Number_of_Stories, Completed_Year)
- region (Region_ID, Name, Capital, Area, Population)

Sub-question 1: Find the Region_ID for the region named "Abruzzo".
Sub-question 2: Filter buildings that belong to the region with the found Region_ID.
Sub-question 3: Retrieve the Name and Number_of_Stories for these buildings.

Divide-and-Merge SQL:

```
SELECT b.Name, b.Number_of_Stories
FROM building b
JOIN (
    SELECT Region_ID
    FROM region
    WHERE Name = 'Abruzzo'
) AS r ON b.Region_ID = r.Region_ID;
```

Extra output: divide-and-merge returns building names in addition to number of stories

Baseline SQL:

```
SELECT b.Number_of_Stories
FROM building b
JOIN region r ON b.Region_ID = r.Region_ID
WHERE r.Name = 'Abruzzo';
```

Figure 7: Failure case with Qwen2.5-32B-Instruct and Qwen2.5-Coder-32B-Instruct using the Last-Sub-query merging strategy. The query asks for the number of stories for each building in the region “Abruzzo.” The Divide-and-Merge SQL correctly filters buildings by region but incorrectly adds building names to the output, returning more information than required. The baseline SQL directly joins `building` and `region` and outputs only `Number_of_Stories`, exactly matching the query intent.

Reasoning Agent	Coding Agent	Merging Strategy	w/o CS	with CS
Qwen2.5-7B-Instruct	Qwen2.5-Coder-7B-Instruct	Last Sub-query	72.26	74.44
		Planner&Executor	66.77	**75.85**
Qwen2.5-14B-Instruct	Qwen2.5-Coder-14B-Instruct	Last Sub-query	69.33	76.61
		Planner&Executor	74.27	**77.16**
Qwen2.5-32B-Instruct	Qwen2.5-Coder-32B-Instruct	Last Sub-query	73.84	78.57
		Planner&Executor	75.58	**79.77**

Table 1: Impact of CS refinement on EX(%) for the Spider test set. Results are shown for different combinations among reasoning agent, coding agent, and merging strategy. Incorporating CS consistently improves performance across settings, with gains of up to 9% compared to models without refinement. The Planner&Executor strategy benefits the most from CS, showing that finer control over column choices enhances alignment with user intent.

Reasoning Agent	Coding Agent	Merging Strategy	AGENTIQL Only	Baseline Only
Qwen2.5-7B-Instruct	Qwen2.5-Coder-7B-Instruct	Last Sub-query	6.96	8.91
		Planner&Executor	7.34	8.21
Qwen2.5-14B-Instruct	Qwen2.5-Coder-14B-Instruct	Last Sub-query	5.00	9.10
		Planner&Executor	4.78	8.91
Qwen2.5-32B-Instruct	Qwen2.5-Coder-32B-Instruct	Last Sub-query	4.94	6.30
		Planner&Executor	4.78	5.54

Table 2: Comparison of instances where AGENTIQL and the baseline model differ in EX on the Spider test set. The columns report proportion of instances where our method succeeds but the baseline fails (**AGENTIQL Only**) and cases where the baseline succeeds but ours fails (**Baseline Only**). Results show that while both methods capture distinct strengths, the Planner&Executor merging strategy reduces the gap relative to the baseline.

Method	Reasoning Agent	Coding Agent	Merging Strategy	Spider-Test
Baseline	Qwen2.5-7B-Instruct	Qwen2.5-Coder-7B-Instruct	-	76.4
Ours			Last Sub-query	74.44
			Planner&Executor	75.85
Baseline	Qwen2.5-14B-Instruct	Qwen2.5-Coder-14B-Instruct	-	80.80
Ours			Last Sub-query	76.61
			Planner&Executor	77.16
Baseline	Qwen2.5-32B-Instruct	Qwen2.5-Coder-32B-Instruct	-	79.93
Ours			Last Sub-query	78.57
			Planner&Executor	79.77
XiYan-SQL [28]	GPT-4o	GPT-4o	-	89.65

Table 3: EX(%) on the Spider test set for the Divide-and-Merge module. Results are reported for Qwen2.5 models of varying sizes. While the baseline achieves strong performance, the Planner&Executor merging strategy improves over the naive last-sub-query approach, demonstrating the benefit of decomposition. Larger models generally yield higher accuracy.

Reasoning Agent	Coding Agent	Merging Strategy	Pearson	Spearmanr
Qwen2.5-7B-Instruct	Qwen2.5-Coder-7B-Instruct	Last Sub-query	-0.61	-0.74
		Planner&Executor	-0.39	-0.79
Qwen2.5-14B-Instruct	Qwen2.5-Coder-14B-Instruct	Last Sub-query	0.46	0.70
		Planner&Executor	0.37	0.52
Qwen2.5-32B-Instruct	Qwen2.5-Coder-32B-Instruct	Last Sub-query	0.35	0.67
		Planner&Executor	0.12	0.64

Table 4: Correlation between schema complexity and performance improvements of our method over the baseline. Schema complexity is measured using a simple metric, which is computed by calculation of the number of tables in the schema. Pearson [29] and Spearman [30] coefficients are reported for different combinations of reasoning agents, coding agents, and merging strategies.

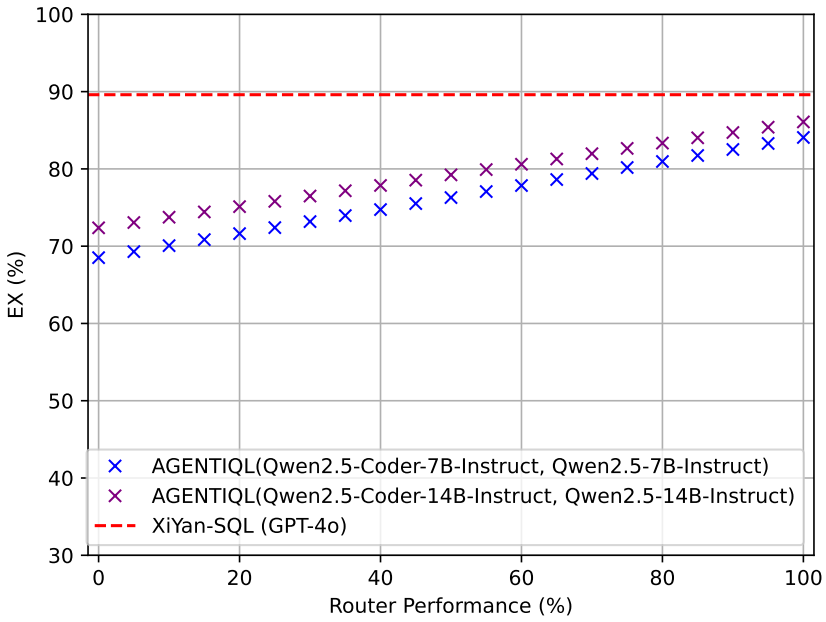

Figure 8: EX(%) on the Spider test set plotted against the adaptive router's accuracy. The curves show AGENTIQL's performance under two configurations (with different model sizes) using the Planner&Executor merging strategy compared to the state-of-the-art system XiYan-SQL. AGENTIQL employs dynamic routing that selects either a modular pipeline (divide-and-merge approach) or a baseline parser for each query. As the router's accuracy increases (i.e., more queries are routed correctly), AGENTIQL's execution accuracy steadily improves, approaching the level of the SOTA system despite using much smaller models. This highlights the effectiveness of dynamic routing in narrowing the performance gap to state-of-the-art solutions while maintaining efficiency.

B Experimental Setup

All local experiments were conducted on an internal compute cluster equipped with NVIDIA A100 GPUs each with 80GB of memory. A total of eight GPUs were available, as confirmed by system diagnostics. For open-source models, we estimate GPU memory requirements based on parameter size: Qwen2.5-7B-Instruct(7B parameters), Qwen2.5-14B-Instruct(14B parameters), Qwen2.5-32B-Instruct (32B parameters), Qwen2.5-Coder-7B-Instruct (7B parameters), Qwen2.5-Coder-14B-Instruct (14B parameters), and Qwen2.5-Coder-32B-Instruct (32B parameters). Some experiments were executed in parallel because they had no dependencies, while others were computed sequentially due to dependency requirements. The total compute estimate for open-source models amounts to approximately 1450 GPU-hours.

C Assets License

We evaluated the following LLMs on the Spider [24] dataset. Below, we list each asset along with its creator and the corresponding license or usage information, where available.

LLMs

- Qwen2.5-7B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link
- Qwen2.5-14B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link

- Qwen2.5-32B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link
- Qwen2.5-Coder-7B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link
- Qwen2.5-Coder-14B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link
- Qwen2.5-Coder-32B-Instruct
  - Creator: Alibaba Cloud
  - License: Apache license 2.0, Hugging Face link
- GPT-4o
  - Creator: OpenAI
  - License: Accessed via API under OpenAI Terms of Use