Supplementary Material

A Effectiveness of QDA-SQL in Data Generation

In this study, QDA-SQL (Questions Enhanced Dialogue Augmentation for Multi-turn Text-to-SQL) is employed to expand the original dataset while adhering to the classification and formatting criteria defined by MMSQL. As shown in Figure 1, QDA-SQL utilizes Chain of Thought (CoT) reasoning to systematically generate multi-turn question-and-answer pairs. Integrating context relationships and varying question types, followed by a refinement process, it guides Gemini Pro in creating diverse datasets that align with our specified question types.

To evaluate the quality of the data generated by QDA-SQL, we performed a manual review of the automatically filtered results, concentrating on both classification accuracy and the quality of the annotated natural language question answering. Our comparison of samples before and after the Verify and Refine process revealed that Gemini Pro successfully identified 94% (47 out of 50) of the misclassified samples. This finding underscores the effectiveness of the automatic filtering process in enhancing data quality and supporting high classification accuracy.

Additionally, an automatic evaluation framework based on GPT-4 was implemented, as outlined in previous research (Zheng et al. 2023; Xu et al. 2023), to assess critical dimensions such as Completeness, Relevance, and Utility in the annotated natural language question answering. To address potential order bias (Iourovitski 2024), the placement of the original dataset and the enhanced dataset was alternated in pairwise comparisons, positioning the enhanced dataset first for odd IDs and second for even IDs. As shown in Figure 2, the enhanced dataset demonstrated superior performance, with an overall assessment indicating that 62% of the QDA-SQL enhanced dataset is considered superior to the manually annotated original SParC and CoSQL training sets. This evidence highlights the effectiveness of QDA-SQL in generating high-quality data.

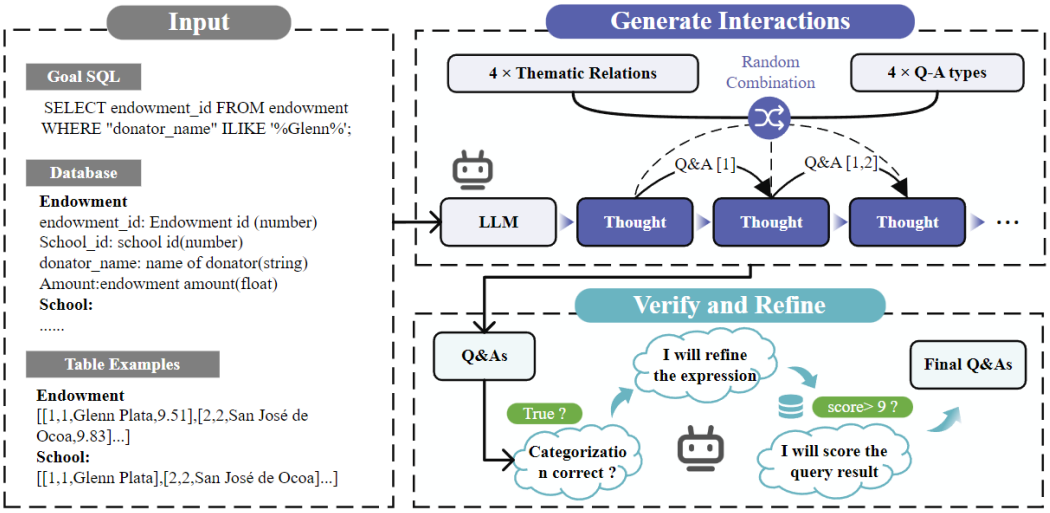

Figure 1: Overview of QDA-SQL processes.

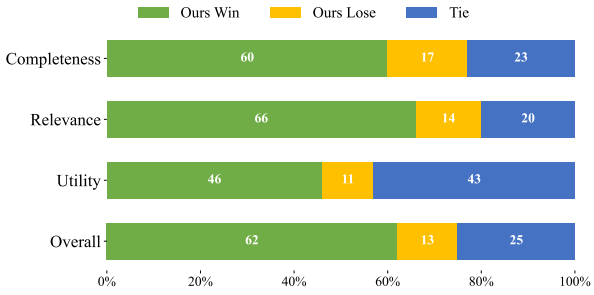

Figure 2: Pairwise comparison of original and QDA-SQL enhanced dataset annotation quality across different criteria.

B The cost analysis of MMSQL test suite

In MMSQL, only the RQS section of the MMSQL testing metrics utilized the OpenAI GPT-4o-mini API through the LLM-assisted rating method. We evaluated four metrics with a scoring range of 0 to 2, leading to a maximum score of 10. This results in API usage costs. Currently, the pricing for GPT-4o-mini is 0.15 $ per million input tokens and 0.60 $ per million output tokens. The scoring cost for our 149 rounds of the test set was estimated to be around 0.036 $, significantly lower than the manual scoring cost, which relies heavily on human labor.

C The Detail of Prompts

Prompt for multi-agent framework

Schema Selector

```
[Instruction] 1. Discard any table schema unrelated to the user question and
evidence. 2. Sort the columns in each relevant table in descending order of
relevance and keep the top 10 columns. 3. Ensure that at least 3 tables are included
in the final output JSON (Contains at least 3 tables which are not "drop_all" even
if it's not very relevant.). 4. The output should be in JSON format.
[Requirements] 1. If a table has less than or equal to 10 columns, mark it as
"keep_all" you don't need to select. 2. If a table is completely irrelevant to the
user question and evidence, mark it as "drop_all". 3. Prioritize the columns in each
relevant table based on their relevance.
==========
Here is a typical example: [DB_ID] car_1 [Schema] {Database Description}{Table Data
Example} [Question] Which companies have three or more models? list the count and
the maker's name. [Evidence] ["'Continent' refers to the continent's Id in table
countries, refers to the name of continent in table continents"] [Answer] The
entities and conditions extracted from the Question are:
entities: count, maker's name
conditions: companies have three or more models
Therefore we can select the related database schema based on these entities with
Evidence.
{"continents": "drop_all", "countries": "drop_all", "car_makers": ["Id", "FullName"],
"model_list": "keep_all", "cars_data": "keep_all"} Question Solved.
==========
Here is a new example, please start answering: [DB_ID] {Database ID} [Schema]
{Database Description}{Table Data Example} [Question] {Conversation History}{Current
Question} [Evidence] {Current Evidence} [Answer] The entities extracted from the
Question are:
```

Question Detector

```
[Requirements] 1. If the user's current question confirms an assumption made in
a previous Answer, the answer will be based on that assumption. You can make
your own limited and reasonable guesses about the user's intent based on the
previous question and the current question. 2. If the user's question is part
of a routine conversation unrelated to the SQL, so the question is improper.
just answer directly. For example, the current question express gratitude or
asks about functions not available outside the database or in llm. don't output
"Yes" but give polite and helpful answers. 3. You need to first try to select
possible corresponding fields for each entity and condition in the question, and
decide whether the question is answerable or ambiguous based on the matches 4.
Determine if the current question was clearly stated but impossible to answer
in SQL based on database schema. If so, the question is unanswerable. Express
Apologies and explain difficulties to the user. You must be absolutely certain
that the question is unanswerable before you can determine that. 5. Check for
ambiguity in the user's current question. If multiple fields have similar meanings
with columns or conditions for user queries the question is ambiguous (Problem is
not enough to generate SQL with sure tables and columns), ask the user to clarify
which field they are referring to or clarify the conditions. 6. If the question
is ambiguous, guess the user's intent and make only minor adjustments to make the
refined question examples answerable and put it in the "rewrite". Choose one from
improper/unanswerable/ambiguous and put it in the "type" 7. Output "answerable" is
```

"Yes" if the question can be answered with certainty for each column and condition. No need to fill in "answer" in this case.
==========
[DB_ID] car_1 [Schema] {Database Description}{Table Data Example} [Question] previous QA: current question: What is the name of AMC? [Evidence] [Answer] The single user question is not routine conversation unrelated to the SQL, not the improper. The question is not related to the current database, not unanswerable. Match possible database sections: entities: name - car_makers.'Maker', car_makers.'FullName', car_names.'Model', car_names'Make' conditions: AMC - model_list.'Model', car_makers.'Maker' Unable to derive user intent through common sense. ask for clarification and give advice and must rewrite an answerable question: {"answerable": "no", "type": "ambiguous", "answer": "Do you mean 'amc' as in model type or carmaker? Did you mean the full name of the carmaker amc?", "rewrite": ["What is the full name of the carmaker 'amc'", "What is the full name of the carmaker which made the model type named 'amc'"]} Question Solved.
==========
Note: Two examples in this prompt are omitted here, representing two scenarios where the question can be answered.
==========
[DB_ID] {Database ID} [Schema] {Database Description}{Table Data Example} [Question] {Conversation History}{Current Question} [Evidence] {Current Evidence}[Answer]

Question Decomposer

Given a [Database schema] description, a knowledge [Evidence] and the [Question], you need to use valid SQLite and understand the database and knowledge, and then decompose the question into subquestions for text-to-SQL generation if the question is complex(If the SQL to be generated is not complex then only one step is needed). When generating SQL, we should always consider constraints: [Constraints] - SELECT Smartly: When writing `SELECT <column>`, only include the columns specifically mentioned in the [Question]. No extras! - FROM & JOIN with Purpose: Don't add tables to `FROM <table>` or `INNER/LEFT/RIGHT JOIN <table>` unless they're absolutely needed for the query. - MAX/MIN Strategy: If you're using `MAX()` or `MIN()`, make sure to do your `JOIN <table>` operations *before* using `SELECT MAX(<column>)` or `SELECT MIN(<column>)`. - Handling "None": If you see 'None' or `None` in the [Value examples] for a column, prioritize using `JOIN <table>` or `WHERE <column> IS NOT NULL` to handle potential missing data effectively. - ORDER BY with GROUP BY: Always include `GROUP BY <column>` before `ORDER BY <column> ASC|DESC` to ensure you're sorting distinct values. - Column Order Matters: The order of columns in your `SELECT` statement should match the order they appear in the question. If the question asks for "count of each of those ... name", your SQL should be `SELECT COUNT(...), name ...` - Counting duplicates: Consider using `DISTINCT` when counting and sorting to avoid counting duplicates. - Fuzzy Text Matching: When you need to match text data in a column and the question suggests finding partial matches, use the '%' wildcard. For example, use `LIKE '%search_term%'` to find values containing "search_term". - Keep it Simple: If the SQL query can be expressed simply and efficiently, don't add unnecessary subquestions. Strive for clarity!
==========
[DB_ID] School [Database schema] {Database Description}{Table Data Example} [Question] List school names of charter schools with an SAT excellence rate over the average. [Evidence] {Current Evidence} Decompose the question into subquestions, considering [Constraints], and generate the SQL after thinking step by step: Subquestion 1: Get the average value of SAT excellence rate of charter schools. SQL ```sql SELECT AVG(CAST(T2.`NumGE1500` AS REAL) / T2.`NumTstTakr`) FROM frpm AS T1 INNER JOIN satscores AS T2 ON T1.`CDSCode` = T2.`cds` WHERE T1.`Charter School (Y/N)` = 1``` Subquestion 2: List out school names of charter schools with an SAT excellence rate over the average. SQL ```sql SELECT T2.`sname` FROM frpm AS T1 INNER JOIN satscores AS T2 ON T1.`CDSCode` = T2.`cds` WHERE T2.`sname` IS NOT NULL AND T1.`Charter School (Y/N)` = 1 AND CAST(T2.`NumGE1500` AS REAL) / T2.`NumTstTakr` > (SELECT AVG(CAST(T4.`NumGE1500` AS REAL) / T4.`NumTstTakr` FROM frpm AS T3 INNER JOIN satscores AS T4 ON T3.`CDSCode` = T4.`cds` WHERE T3.`Charter School (Y/N)` = 1)``` Question Solved.

==========
[DB_ID] Salary [Database schema] {Database Description}{Table Data Example} [Evidence] {Current Evidence} [Question] What is the gender of the youngest client who opened account in the lowest average salary branch? Decompose the question into subquestions, considering [Constraints], and generate the SQL after thinking step by step: The ordered presentation columns of the final SQL are as follows: 1. gender Sub question 1: What is the district_id of the branch with the lowest average salary? SQL ```sql SELECT `district_id` FROM district ORDER BY `A11` ASC LIMIT 1``` Sub question 2: What is the youngest client who opened account in the lowest average salary branch? SQL ```sql SELECT T1.`client_id` FROM client AS T1 INNER JOIN district AS T2 ON T1.`district_id` = T2.`district_id` ORDER BY T2.`A11` ASC, T1.`birth_date` DESC LIMIT 1``` Sub question 3: What is the gender of the youngest client who opened account in the lowest average salary branch? SQL ```sql SELECT T1.`gender` FROM client AS T1 INNER JOIN district AS T2 ON T1.`district_id` = T2.`district_id` ORDER BY T2.`A11` ASC, T1.`birth_date` DESC LIMIT 1``` Question Solved.
==========
[DB_ID] {Database ID} [Schema] {Database Description}{Table Data Example} [Question] {Conversation History}{Current Question} [Evidence] {Current Evidence} Decompose the question into subquestions, considering [Constraints], and generate the SQL after thinking step by step:

SQL Refiner

[Instruction] When executing SQL below, some errors occurred, please fix up SQL based on query and database info. Solve the task step by step if you need to. Using SQL format in the code block, and indicate script type in the code block. When you find an answer, verify the answer carefully. Include verifiable evidence in your response if possible. Only SQL statements are allowed in (the fixed SQL), do not add any comments. [Constraints] - In `SELECT <column>`, just select needed columns in the [Question] without any unnecessary column or value - In `FROM <table>` or `JOIN <table>`, do not include unnecessary table - If use max or min func, `JOIN <table>` FIRST, THEN use `SELECT MAX(<column>)` or `SELECT MIN(<column>)` - If [Value examples] of <column> has 'None' or None, use `JOIN <table>` or `WHERE <column> is NOT NULL` is better - If use `ORDER BY <column> ASC|DESC`, add `GROUP BY <column>` before to select distinct values [Response format] Your response should be in this format: Analysis: **(Your analysis)** Correct SQL: ```sql (the fixed SQL) ``` [Query] {Conversation History}{Current Question} [Evidence] {Current Evidence} [Database] {Database Description}{Table Data Example} [old SQL] {Wrong SQL} [SQLite error] {Error Log} Now please fix up old SQL and generate new SQL again.

Prompt for RQS metric

The RQS metric evaluates the quality of natural language responses across five dimensions: Utility, Accuracy, Completeness, Clarity, and Relevance. For ease of reference, the evaluation criteria are detailed in Table 1. The simplified prompt is as follows:

Evaluate the quality of the system's response based on the following criteria. Assign 2 points directly if a criterion does not apply. {Detailed Criteria} Task: Classify the Response: Determine if the system response is 'improper'(Non-SQL based user questions), 'unanswerable'(unachievable under existing conditions), or 'ambiguous'(Lack of clarity). Evaluate Each Criterion: Provide a detailed rationale for the score assigned to each criterion. Calculate the Total Score: Sum the scores for all criteria.(10 points for a direct greeting alone) Output Format: {{ "AnswerType": "",(text only) "Rationale": "",(text only, Explain the scoring of each criterion) "Score": ""(An integer from 0 to 10) }}

Prompt for QDA-SQL

Chain-of-Thought

Based on {Conversation History} {Current Q-A Type Requirement} {Current Thematic Relation Requirement} {Database Description} {Table Data Example} Based on the above content, referring to {Goal SQL} generate a new dialogue based on the previous conversation. The newly generated SQL can differ from the target SQL. The question you ask needs to be difficult enough that it requires the use of

Criterion	Score	Description
Relevance	0	The response is completely irrelevant.
	1	The response is partially relevant but misses key details.
	2	The response is fully relevant and adequately addresses the question.
Clarity	0	The response is incomprehensible.
	1	The response is mostly clear with minor ambiguities.
	2	The response is very clear and easy to understand.
Completeness	0	The response does not address the question at all.
	1	The response covers most aspects of the question but lacks some details.
	2	The response thoroughly addresses all aspects of the question.
Accuracy	0	The response contains factually incorrect information.
	1	The response is partially accurate with some errors.
	2	The response is completely accurate.
Utility	0	The response does not meet the user's needs or explain the context of the question.
	1	The response somewhat meets the user's needs and provides partial explanations.
	2	The response excellently meets the user's needs and clearly explains the context or ambiguity of the question.

Table 1: Evaluation Criteria for Response Quality

`complex SQL to answer it. Output 5 questions and corresponding SQL in the format: [{text:"",sql:""}, {text:"",sql:""}, ...] Output only in JSON format.`

Note: For answerable types of questions, the LLM needs to generate 5 such questions along with their corresponding SQL answers simultaneously. These answers will be validated through database execution. Only those answers that are executable and yield non-empty results will be retained to ensure that the generated samples contain more meaningful Q&A.

Verifying Alignment

`Based on {Conversation History} {Database Description} {Table Data Example} {Q-A Types Definition} {Multi-turn Dialogue Sample}[-2] and {Multi-turn Dialogue Sample}[-1] Based on the above content, is the classification correct? (For Unanswerable: can this question be achieved with more complex SQL or multi-table join query for valid SQL?) (For Ambiguous: To make sure that the question is indeed impossible to answer directly based on the information above) If correct, output {"type": "yes"} otherwise {"type": "no"} Output nothing else except JSON.`

Optimizing Expression

`Based on {Database Description} {Table Data Example} {Q-A Types Definition} {expression and Context Coherence Requirements:} Modify the {Multi-turn Dialogue Sample} based on the above requirements, and output in the same format after revision.`

Scoring SQL Execution

`Based on {Conversation History} {Database Description} {Current Question} {SQL answer to Current Question} {SQL Execution Result} Does the SQL and its execution result fully meet the user's needs? Output the rating of the system's response (1-10), output the rating result {"score":""} Output nothing else except JSON.`

D Impact of fine-tuning

As discussed, our study has highlighted advancements in using LLMs for text-to-SQL applications, yet it also reveals several limitations that point to opportunities for further development. Our multi-agent framework and experiments are currently based on LLMs that have not undergone further fine-tuning. Exploring open-source, domain-specific LLMs and further decomposing and annotating the MMSQL training set could enhance our framework's effectiveness synergistically.

In this context, we investigated the impact of fine-tuning on the relatively smaller open-source LLM, CodeLlama-7B. Despite the lack of detailed annotations necessary for a multi-agent framework, we utilized the MMSQL dataset to perform simple fine-tuning in a zero-shot setting. As shown in Table 2, the fine-tuning process significantly enhanced the model's performance across all metrics. We randomly selected 50 records pre- and post-fine-tuning to assess the impact further. We manually counted the

number of errors, categorizing them into four types: Semantic Understanding Errors, SQL Execution Errors, Logical Errors, and Database Comprehension Errors. The fine-tuned model exhibited a reduction in all four types of errors, with the total error count dropping from 35 to 15. Notably, the model significantly reduced errors in the Database Comprehension and Logical categories, decreasing from 12 to 5 and 14 to 6, respectively. See Appendix E for detailed error analysis.

These findings indicate that fine-tuning enhances the model's understanding of domain-specific issues and improves its ability to generate SQL and natural language, achieving performance comparable to closed-source models in a zero-shot setting. This demonstrates the potential of this approach.

Table 2: Comparison of model performance before and after fine-tuning

Model	TDEX	EX	Average RQS	F1 Score	Err. Count (50 records)
GPT-4 Turbo	67.0	70.0	5.80	68.2	21
Codellama-7B Base	30.7	27.2	5.09	46.9	35
Codellama-7B SFT	64.1	67.5	5.20	57.8	15

E Error analysis for different approaches

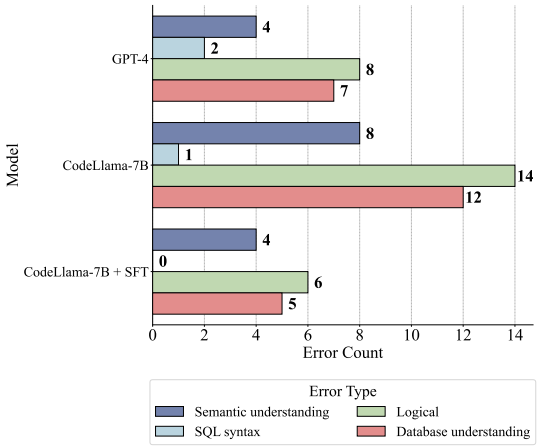

Figure 3: Error Type Distribution Across Different Models.

In our analysis, We categorized error reasons into the following classes:

- **Semantic understanding errors**: These include insufficient contextual understanding, misinterpretation of the question, and word confusion.
- **SQL execution errors**: These arise from issues such as incorrect function usage and data mismatches, leading to execution failures.
- **Logical errors**: These involve incorrect relational expressions, conditional judgments, and aggregate operations errors.
- **Database comprehension errors**: These include unfamiliarity with the database structure, limitations related to database functions, and incorrect selection of relevant tables.

We manually classified errors in the outputs of the three models, acknowledging that a single response could contain multiple errors. Figure 3 illustrates the error distribution.

References

Iourovitski, D. 2024. Grade Score: Quantifying LLM Performance in Option Selection. arXiv:2406.12043.

Xu, C.; Sun, Q.; Zheng, K.; Geng, X.; Zhao, P.; Feng, J.; Tao, C.; and Jiang, D. 2023. WizardLM: Empowering Large Language Models to Follow Complex Instructions. arXiv:2304.12244.

Zheng, L.; Chiang, W.-L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; Zhang, H.; Gonzalez, J. E.; and Stoica, I. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 46595–46623. Curran Associates, Inc.