

Reliable Text-to-SQL in Resource-Constrained Settings with Small LLMs via Iterative Retry Workflows

Bilel Abderrahmane Benziane¹ and Xavier Schepler¹

¹Data Lab, Generix Group, Paris, France

Emails: babenziane@generixgroup.com, xschepler@generixgroup.com

Abstract—Large Language Models (LLMs) have recently improved Text-to-SQL generation, but their computational cost can hinder deployment in resource-constrained settings. This paper evaluates the Text-to-SQL capabilities of small and medium open-source LLMs (1B–8B parameters) under a controlled inference budget, focusing on how model specialization and prompt formulation affect performance. Using the Spider benchmark, we compare multiple prompt representations and assess the accuracy of generated queries. Our results show that SQL-specialized models are consistently more reliable than general-purpose and chat-oriented models at small scales, and that performance can vary substantially depending on the prompt used. To reduce this variability without resorting to larger models, we introduce three lightweight inference workflows that automatically retry with alternative prompts and/or models when failures occur, including an optional syntax-correction step. These workflows significantly improve end-to-end success rates, with a trade-off in additional inference calls. These results demonstrate a practical path to stronger Text-to-SQL performance with small LLMs, by trading modest additional inference cost for higher reliability.

Index Terms—Text-to-SQL, Large Language Models, Resource-Constrained Inference, Prompting, Spider Benchmark

I. INTRODUCTION

With the abundance of data in the last few years, artificial intelligence has witnessed rapid growth in the use of foundation models [1]. Training a model on a wide range of data can help improve generalization and offer better inference on unseen data. One AI subfield that has witnessed a rapid increase in performance is natural language processing, where the use of large language models (LLMs) has rapidly increased in the last few years [2], [3].

However, a major limitation of LLMs is their lack of knowledge on specific tasks. To increase the performance of an LLM, it has to be tuned on a specific type of data or field [4]. As a result of fine-tuning LLMs, many specialized models have emerged in the last few years, such as instruction-tuned/chat LLMs, code LLMs, and domain-specific LLMs [5]–[9]. One special type of code LLM is Text-to-SQL. The importance of these specialized models specifically lies in their direct impact on democratizing data access by offering a means for people with a non-technical background to query complex databases.

Although LLMs are important in facilitating tasks, they tend to be resource-hungry. Unlike traditional smaller models, LLMs require much more expensive resources [10]. This could lead to an exponential increase in the business cost of any

given software solution making use of an LLM. Previous work tackled this problem by compressing the models to lower numerical precision. One example is the Q8BERT model, an 8-bit quantized version of BERT obtained via quantization-aware fine-tuning, achieving roughly $4\times$ compression and potential inference speedups with negligible accuracy loss [11]. While these methods reduce memory use, they degrade performance, usually require tuning quantization further after training, and have only been studied for models with fewer than 350M parameters.

Previous research tackled the same problem on a larger scale by carrying out more extensive experimentation on large and medium-sized LLMs [12]. The results showcased a positive correlation between model scale and performance, where the larger version of fine-tuned LLaMA-2 (CodeLLaMA-34B) achieved an average execution accuracy of 39.6% compared to the smaller version (LLaMA-2-CHAT-13B), which achieved an average execution accuracy of 19.4%. This work also showed that question representation plays an important role in improving performance. Building on previous work, this study focuses on assessing the performance of medium and small LLMs for the Text-to-SQL task. In order to improve the performance of smaller LLMs, this work suggests three different workflows to improve accuracy.

The remainder of this paper is organized as follows. Section II presents the related work. Section III describes the proposed methodology. Section IV reports and discusses the experimental results. Finally, Section V concludes this work and outlines possible directions for future research.

II. RELATED WORK

Many paths exist in the literature to improve the performance of LLMs. In the next sections, two main techniques are discussed.

A. Fine Tuning

Fine-tuning a generic LLM is the first step toward obtaining a specialized Text-to-SQL model. For instance, Qwen2.5-Coder-7B-Instruct results from training Qwen2.5 on a large corpus of programming languages and not only SQL [13]. Deepseek-ai/deepseek-coder-6.7b-instruct is a tuned version of deepseek-ai/deepseek-coder-6.7b-base [14]. Similarly, codellama/CodeLlama-7b-Instruct-hf is the

instruction-tuned variant of Code Llama 7B, a code-specialized model obtained by further training Llama 2 on code-centric data and then aligning it for instruction following [15].

These models have between 6 and 8 billion parameters, which is considered medium-sized. Smaller models with as few as 1 billion parameters have appeared recently. These small models are not only code-specialized but also SQL-specialized. One example is `seeklhy/codes-1b-spider`, which is fine-tuned using a curated SQL-centric corpus [16]. Another recently released SQL generation model is `prem-research/prem-1B-SQL`. This model is based on DeepSeek-Coder-1.3B-Instruct [14]. Due to the abundance of open-sourc fine-tuned code models, this work focuses on evaluating already pre-trained models.

B. Prompt Styles

Another part of the literature focuses on prompt engineering for the query generation process. Any given question can be enriched with extra context before being passed to the model to ensure that it has enough information for SQL generation. This process is known as question representation. The most basic type of question representation is called the Basic Prompt [17]. This representation includes each table name and columns, followed by the prefix “Q:” and the original natural language question, and an answer prefix such as “A: SELECT” to indicate that the output should be an SQL query.

Another type of prompt is the Text Representation [18]. This prompt is similar to the Basic Prompt, with an additional instruction at the beginning to make the task clearer. OpenAI released their own prompt for this task, which further clarifies the format by adding the symbol “#” to information fields and explicitly asking the model to complete the SQL query without explanation [19]. Another suggested prompt is Alpaca SFT [20]. This prompt is very similar to the OpenAI prompt, but the instruction is more detailed, as it was originally designed for fine-tuning purposes.

A common limitation of the aforementioned prompting styles is the lack of full table schemas, which is crucial for generating SQL queries that join multiple tables. For that purpose, the Core Representation Prompt adds the full SQL schema for each table to the question [21]. This includes foreign keys, which are important for generating multi-table queries.

Despite the abundance of papers on LLMs, evaluations of medium and small LLMs for SQL generation remain scarce. To fill this research gap, this work focuses on evaluating multiple well-established LLMs for the Text-to-SQL task across different prompting styles and SQL query types. In order to improve the performance of small LLMs, three SQL generation workflows are suggested and evaluated against a baseline.

III. METHODOLOGY

This section presents the experimental setup used in this work. We describe the dataset, the evaluation metrics, the set

of models considered, the prompting configurations used for inference, and the proposed workflows designed to improve robustness in resource-constrained settings.

A. Dataset

For evaluation, we use the Spider dataset [22]. Spider contains 10,181 cross-domain natural language questions collected from 200 databases, paired with corresponding SQL queries. Following the experimental setup suggested by [12], we use the official Spider development partition, comprising the last 1,034 examples, as the evaluation set. Figure 1 illustrates the distribution of query types in the evaluation data: simple SELECT, GROUP BY, ORDER BY, JOIN, AGGREGATE, SUBQUERY, and HAVING.

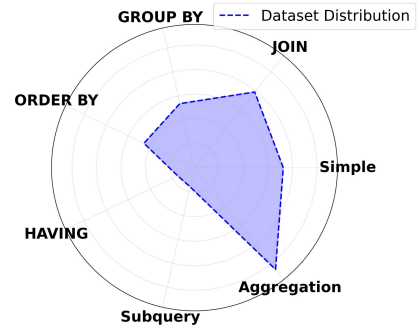


Fig. 1: Distribution of SQL query types in the Spider evaluation set.

B. Metrics

To make a fair comparison, we follow the evaluation protocol used in prior work [12]. To evaluate performance, we use exact-set-match accuracy (EM) and execution accuracy (EX). EM checks whether the predicted SQL query exactly matches the ground-truth query at the clause level (i.e., set-level comparison of keywords and predicates), making it a conservative metric: two semantically equivalent queries that differ in form will not match. EX, by contrast, compares the execution results of the predicted and ground-truth queries on the target database. Because a single question can often be answered by multiple semantically equivalent SQL queries, EX typically provides a more faithful estimate of model performance.

C. Models

To study how model specialization affects Text-to-SQL performance, we evaluate models from multiple families. Specifically, the tested models are grouped into five categories: (i) *fundamental* (base) LLMs, (ii) *instruction-tuned* LLMs, (iii) *code* LLMs, (iv) *chat* LLMs, and (v) *SQL-specialized* LLMs.

Fundamental models. We evaluate base checkpoints including Llama-3-8B [23], Qwen2.5-7B [24], Mistral-7B [25], DeepSeek-LLM-7B-base [26], and TinyLlama-1.1B [27].

Instruction-tuned models. We also test instruction-tuned variants, including Llama-3-8B-Instruct [23], Qwen2.5-7B-Instruct [24], and Mistral-7B-Instruct [25].

Structured Prompt

```

### Instruction:
Translate the following question into SQL.

### Database Schema:
continents (ContId, Continent)
countries (CountryId, CountryName, Continent)

### Question:
How many continents are there ?

### SQL:
SELECT

```

Fig. 2: Structured prompt template for Text-to-SQL inference.

Code models. For code-oriented generation, we evaluate Qwen2.5-Coder-7B and its instruction-tuned variant [13], CodeLlama-7B and CodeLlama-7B-Instruct [15], DeepSeek-Coder-6.7B base and instruct [14], StarCoder2-3B [28], and CodeS-1B [16].

Chat models. To represent chat-optimized checkpoints, we include DeepSeek-LLM-7B-Chat [26], Vicuna-7B v1.5 [29], Llama-2-7B-Chat [30], and TinyLlama-1.1B-Chat [27].

SQL-specialized models. Finally, we test models explicitly targeted at Text-to-SQL, including Defog-SQLCoder-7B [31], Prem-Text-to-SQL-1B [32], and CodeS variants fine-tuned for Text-to-SQL, including CodeS-1B-spider and CodeS-1B-bird [16].

To ensure a fair comparison across methods, we allocate 150 tokens for response generation for all models. By default, we set the temperature to 0 to eliminate the influence of randomness. Each model is tested using the prompt styles discussed in Section II. In addition, we introduce a *structured prompt* (Fig. 2) that organizes the input into explicit sections (instruction, database schema, and question). This prompt follows a structured format while using a simplified schema description, enabling a more consistent comparison across prompt styles.

D. Proposed Workflows

Model performance may vary with both the question type and the prompt representation. As a result, using a single model and a single prompt style in an inference system can lead to a higher number of errors. To improve robustness, we propose three workflows. **Single-Model/Prompt Baseline.** This algorithm defines the baseline setting in which both the prompt style P and the model M are fixed. Given a question q , the system constructs a prompt p using P , generates an SQL query $\hat{s}$ using M , and returns the execution result R if $\hat{s}$ executes successfully.

1) *Prompt Workflow:* In the prompt workflow (Algorithm 1), the model M is fixed and the system iterates over a list of prompt styles $P = [P_1, \dots, P_K]$. For each P_i , the system generates a candidate query $\hat{s}_i$ and attempts execution. If execution fails due to an SQL validation error, the same model M is used once to produce a corrected query $\tilde{s}_i$ conditioned on $(q, \hat{s}_i)$. The workflow returns the first result R obtained from a valid query; otherwise, it proceeds to P_{i+1} until all prompt styles are exhausted.

Algorithm 1 Prompt-Flow with Syntax Correction

```

1: Input: Question  $q$ , list of prompt styles  $P = [P_1, \dots, P_K]$ , model  $M$ 
2: Output: Query result set  $R$  or failure
3: Define prompt styles list  $P$ 
4: Define Text-to-SQL model  $M$ 
5: Read question  $q$ 
6:  $i \leftarrow 1$ 
7: while  $i \leq K$  do
8:   Format  $q$  using prompt style  $P_i$  to obtain prompt  $p_i$ 
9:   Pass  $p_i$  to model  $M$  to generate SQL query  $\hat{s}_i$ 
10:  Execute  $\hat{s}_i$  on the database
11:  if no SQL validation error then
12:    Obtain result  $R$ 
13:    return  $R$ 
14:  else
15:    Pass  $\hat{s}_i$  and  $q$  to  $M$  for syntax correction to obtain  $\tilde{s}_i$ 
16:    Execute  $\tilde{s}_i$  on the database
17:    if no SQL validation error then
18:      Obtain result  $R$ 
19:      return  $R$ 
20:    else
21:       $i \leftarrow i + 1$ 
22:    end if
23:  end if
24: end while
25: return failure

```

▷ All prompt styles exhausted with invalid SQL

2) *Model Workflow:* In the model workflow (Algorithm 2), the prompt style P is fixed and the system iterates over a list of models $M = [M_1, \dots, M_K]$. For each M_i , the system generates a candidate query $\hat{s}_i$ from the same prompt p and attempts execution. If execution fails due to an SQL validation error, the same model M_i is used once to generate a corrected query $\tilde{s}_i$ conditioned on $(q, \hat{s}_i)$. The workflow returns the first result R obtained from a valid query; otherwise, it proceeds to the next model.

Algorithm 2 Model-Flow with Syntax Correction

```

1: Input: Question  $q$ , prompt style  $P$ , list of models  $M = [M_1, \dots, M_K]$ 
2: Output: Query result set  $R$  or failure
3: Define prompt style  $P$ 
4: Define Text-to-SQL models list  $M$ 
5: Read question  $q$ 
6:  $i \leftarrow 1$ 
7: while  $i \leq K$  do
8:   Format  $q$  using prompt style  $P$  to obtain prompt  $p$ 
9:   Use model  $M_i$  to generate SQL query  $\hat{s}_i$  from  $p$ 
10:  Execute  $\hat{s}_i$  on the database
11:  if no SQL validation error then
12:    Obtain result  $R$ 
13:    return  $R$ 
14:  else
15:    Pass  $\hat{s}_i$  and  $q$  to  $M_i$  to obtain corrected SQL  $\tilde{s}_i$ 
16:    Execute  $\tilde{s}_i$  on the database
17:    if no SQL validation error then
18:      Obtain result  $R$ 
19:      return  $R$ 
20:    else
21:       $i \leftarrow i + 1$ 
22:    end if
23:  end if
24: end while
25: return failure

```

▷ All models used and SQL remains invalid

3) *Prompt-and-Model Workflow:* In the prompt-and-model workflow (Algorithm 3), the system iterates over combinations of prompt styles and models. Given prompt styles $P = [P_1, \dots, P_K]$ and models $M = [M_1, \dots, M_L]$, the workflow loops over models M_j and, for each M_j , loops over prompt styles P_i . For each pair (M_j, P_i) , the system generates $\hat{s}_{i,j}$, attempts execution, and, upon validation failure, applies a

single correction step using the same model M_j to produce $\tilde{s}_{i,j}$ conditioned on $(q, \hat{s}_{i,j})$. The workflow returns the first valid result R ; if all pairs fail, it returns failure.

Algorithm 3 Model-and-Prompt Flow with Syntax Correction

```

1: Input: Question  $q$ , prompt styles list  $P = [P_1, \dots, P_K]$ , models list  $M = [M_1, \dots, M_L]$ 
2: Output: Query result set  $R$  or failure
3: Define prompt styles list  $P$ 
4: Define Text-to-SQL models list  $M$ 
5: Read question  $q$ 
6:  $j \leftarrow 1$  ▷ model index
7: while  $j \leq L$  do
8:    $i \leftarrow 1$  ▷ prompt-style index
9:   while  $i \leq K$  do
10:    Format  $q$  using prompt style  $P_i$  to obtain prompt  $p_{i,j}$ 
11:    Use model  $M_j$  to generate SQL query  $\hat{s}_{i,j}$  from  $p_{i,j}$ 
12:    Execute  $\hat{s}_{i,j}$  on the database
13:    if no SQL validation error then
14:      Obtain result  $R$ 
15:      return  $R$ 
16:    else ▷ Syntax correction with same model
17:      Pass  $(q, \hat{s}_{i,j})$  to  $M_j$  for syntax correction to obtain  $\tilde{s}_{i,j}$ 
18:      Execute  $\tilde{s}_{i,j}$  on the database
19:      if no SQL validation error then
20:        Obtain result  $R$ 
21:        return  $R$ 
22:      else
23:         $i \leftarrow i + 1$  ▷ Try next prompt style
24:      end if
25:    end if
26:  end while
27:   $j \leftarrow j + 1$  ▷ All prompts tried, move to next model
28: end while
29: return failure ▷ All  $(M_j, P_i)$  combinations failed

```

IV. RESULTS AND DISCUSSION

This section reports the performance of the evaluated models across the different prompt styles. Table I summarizes execution accuracy (EX) and exact-set-match accuracy (EM) for each model and prompt style.

Model families. Overall, SQL-specialized models achieve the strongest results, with all models in this group reaching an average execution accuracy above 55%. Code-oriented models generally follow, but their performance is more sensitive to the prompt style. For example, CodeLlama-7B, its instruction-tuned variant, StarCoder2, and DeepSeek-Coder show strong variations across prompt formats, performing well only under certain representations. The instruction-tuned DeepSeek-Coder variant is more robust with Alpaca-style prompts, suggesting better alignment with its training format. Notably, CodeS achieves high accuracy across most prompt styles despite having only 1B parameters. General-purpose instruction-tuned models, such as Qwen2.5-7B-Instruct and Llama-3-8B-Instruct, underperform their code-tuned counterparts, confirming the importance of code and task-oriented fine-tuning. Fundamental models perform worse overall, while chat-oriented models are the weakest family, with average execution accuracy below 30%.

Model size. This study targets reliable Text-to-SQL generation with smaller models. While results vary across families, the 1B-parameter models show clear differences: SQL-specialized 1B models reach average execution accuracy above 55%, followed by the code-oriented 1B model at approximately 36%, then chat-oriented models at approximately 18%,

and finally fundamental models at approximately 4%. These results support two main observations: (i) SQL-specific fine-tuning is particularly important at small model scales, and (ii) prompt choice remains a major factor in achieving strong performance.

Prompt style. Beyond model family, prompt style has a strong impact on performance. The results show that the same model can behave very differently depending on how the schema and question are presented. To analyze this effect, we compare prompt styles across SQL query categories for the 1B SQL-oriented models, which are the main focus of this work in resource-constrained settings. In Fig. 3(a), the green polygon represents EM and the blue polygon represents EX, while each radar axis corresponds to a query category. As shown in Fig. 3(a), performance varies not only by prompt style but also by query type. In particular, the code representation prompt performs better on multi-table queries, likely because it provides richer schema information. Overall, no single prompt style is uniformly optimal across models and query categories, making prompt selection an important component of a Text-to-SQL system.

Based on this observation, we evaluate the three proposed workflows introduced in Section III-D. The goal is to improve robustness by adapting the inference strategy after a failed attempt, either by changing the prompt style, switching to another model, or combining both. Table II summarizes the accuracy of these workflows.

The prompt-flow workflow is evaluated with both Prem-1B-SQL and CodeS-1B-Spider. In both cases, it improves execution accuracy compared to any single prompt style, showing that varying P after failure can substantially reduce errors. By contrast, model-flow generally provides smaller gains, suggesting that changing M alone is not sufficient when prompt formatting is a major source of failure. Finally, the model-and-prompt workflow reaches the same execution accuracy as the best prompt-flow configuration, but requires more inference calls, making it less suitable for resource-constrained settings.

Overall, prompt-flow offers the best accuracy–cost trade-off in our setup: it improves robustness while keeping the number of additional inference calls moderate. When maximum accuracy is the main objective, model-and-prompt flow can further reduce failures, but with higher computation. Thus, all workflows trade additional inference cost for higher reliability, and the best choice depends on the deployment constraints.

Fig. 3 compares the baseline and the three proposed workflows across query categories. Compared to the baseline in Fig. 3(a), the workflows generally expand the EX profile (blue), indicating fewer execution failures. Model-Flow in Fig. 3(b) gives moderate improvements, while Prompt-Flow in Fig. 3(c) provides more consistent gains, showing that adapting P after failure is effective for small Text-to-SQL models. Finally, Model-and-Prompt Flow in Fig. 3(d) achieves the strongest overall profile, but at the cost of more inference calls, consistent with Table II.

Type	Model ID	Alpaca Sft		Basic Prompt		Code Repr.		OpenAI Demo		Struct. Prompt		Text Repr.		Avg	
		EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX
Fund.	Llama-3-8B	21.86	70.50	2.32	33.08	28.53	3.10	29.21	6.29	25.15	74.95	25.05	39.27	22.02	37.86
	Qwen2.5-7B	31.43	87.81	4.45	27.66	39.27	43.42	37.91	30.37	37.23	83.08	37.91	90.43	31.37	60.46
	Mistral-7B	21.95	27.27	2.03	27.47	32.01	17.70	29.01	0.19	3.58	0.00	31.43	0.19	20.00	12.14
	Llama-2-7B	9.67	0.77	0.77	16.83	14.99	1.06	12.19	0.77	0.00	0.00	12.09	0.58	8.29	3.34
	Deepseek-7B	16.34	0.77	2.71	26.50	23.99	0.58	21.57	0.19	20.02	0.00	20.21	0.10	17.47	4.69
	TinyLlama-1.1B	3.97	0.29	1.64	22.82	13.44	0.87	10.64	0.48	5.61	0.10	7.83	0.10	7.19	4.11
Instr.	Llama-3-Inst	16.63	0.00	0.00	13.83	23.21	3.58	34.24	0.00	0.00	0.00	20.50	0.00	15.76	2.90
	Qwen2.5-Inst	27.27	0.10	2.42	7.25	42.17	88.01	39.85	58.90	0.10	0.00	42.26	28.05	25.68	30.38
	Mistral-Inst	11.51	63.93	0.00	9.28	26.98	82.21	25.44	15.38	0.77	2.61	23.69	5.80	14.73	29.87
Code	Qwen2.5-Coder	31.82	84.72	8.90	41.49	40.33	36.27	40.72	89.56	40.14	89.56	41.39	91.20	33.88	72.13
	Qwen2.5-Coder-I	35.78	89.94	0.48	1.45	49.61	94.97	54.26	93.33	3.19	10.45	46.62	58.41	31.66	58.09
	CodeLlama-7b	19.15	0.39	3.77	26.02	31.43	0.58	26.98	0.00	0.77	0.00	24.37	0.29	17.75	4.55
	CodeLlama-Inst	22.82	28.72	1.26	23.31	33.75	0.77	26.69	4.26	23.02	3.19	29.40	0.29	22.82	10.09
	Deepseek-Coder	35.59	7.16	4.26	30.08	39.75	0.77	41.20	6.77	39.56	88.59	38.59	3.29	33.16	22.78
	Deepseek-Coder-I	19.15	51.64	0.10	22.05	36.27	1.06	39.85	0.00	15.67	0.00	37.43	3.00	24.74	12.96
	StarCoder2-3B	22.44	0.00	9.57	45.75	32.30	0.87	30.27	0.00	2.52	0.00	31.04	0.00	21.36	7.77
	CodeS-1B	8.90	52.22	7.93	40.72	31.14	0.10	23.69	0.29	20.02	72.44	23.31	55.71	19.17	36.91
Chat	DeepSeek-Chat	17.31	63.64	0.29	10.83	28.72	45.16	29.59	40.62	5.03	0.87	27.95	14.12	18.15	29.21
	Vicuna-7b	5.22	14.22	0.39	4.35	11.03	22.73	12.28	27.76	2.42	7.16	5.51	16.54	6.14	15.46
	Llama-2-Chat	13.54	0.10	0.19	14.12	18.67	0.10	14.31	6.77	2.22	0.19	10.83	0.19	9.96	3.58
	TinyLlama-Chat	4.74	10.93	0.39	12.77	15.57	40.72	12.09	9.57	9.28	19.92	10.64	19.92	8.79	18.97
SQL	SQLCoder-7b	0.87	80.17	0.00	0.00	0.29	77.37	1.45	77.18	0.58	75.44	0.58	74.86	0.63	64.17
	Prem-1B-SQL	6.19	71.66	0.39	18.18	56.67	7.74	42.75	87.04	40.43	88.69	42.65	60.06	31.51	55.56
	CodeS-1B-Spider	27.85	65.96	42.55	74.57	41.20	77.85	42.36	72.82	41.20	71.95	41.10	71.95	39.38	72.52
	CodeS-1B-Bird	15.28	69.54	21.66	64.31	20.89	73.31	19.05	69.15	21.18	69.54	21.86	69.15	19.99	69.17

TABLE I: Model performance (EM/EX) across prompt styles on the Spider evaluation set.

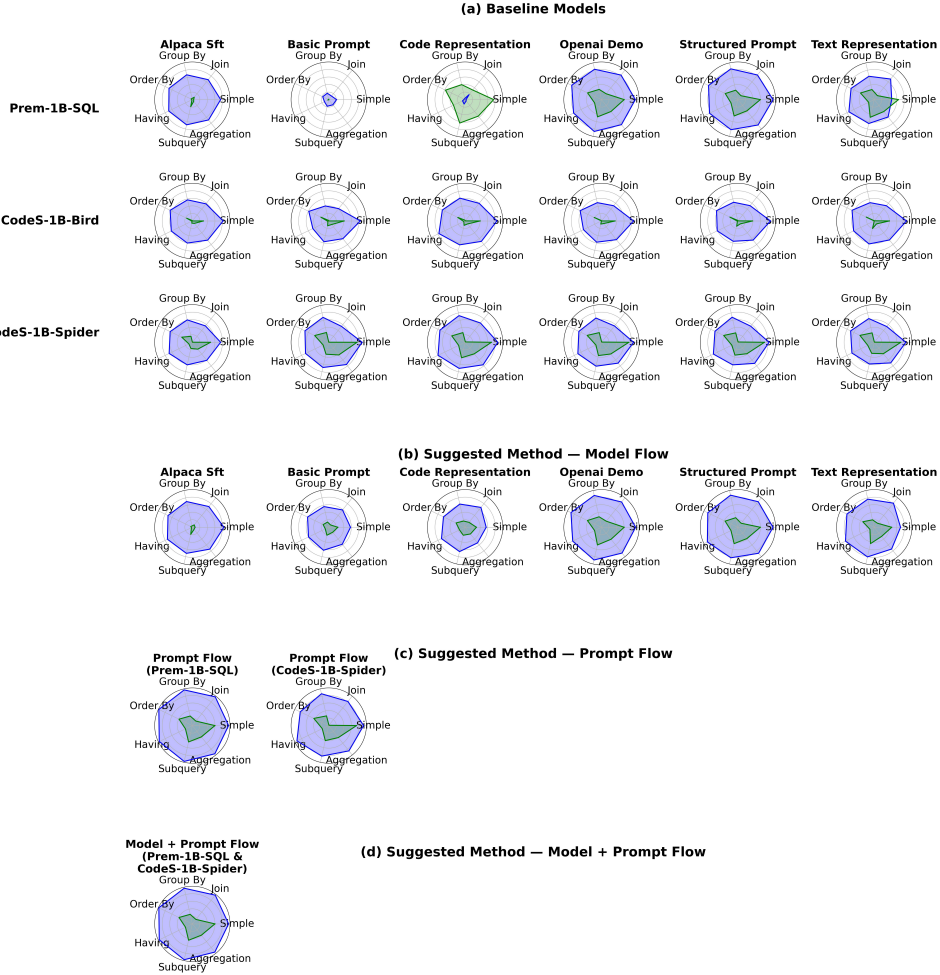


Fig. 3: Radar plots of EM (green) and EX (blue) across query categories. (a) Baseline single-pass inference with different prompt styles. (b) Model-Flow workflow. (c) Prompt-Flow workflow. (d) Model-and-Prompt Flow workflow.

TABLE II: Workflow Accuracy Comparison (EX / EM)

Algorithm	Repr	EX	EM
Prompt workflow (Prem-1B-SQL)	–	97.20	41.30
Prompt workflow (CodeS-1B-Spider)	–	87.81	43.13
Model and prompt workflow	–	97.20	41.30
Model workflow	Alpaca Sft	76.69	7.83
	Basic Prompt	58.99	16.34
	Code Repr.	62.96	23.89
	OpenAI Demo	88.78	43.13
	Struct. Prompt	90.33	40.62
	Text Repr.	77.27	33.46

V. CONCLUSIONS

This work investigated a range of LLM families for the Text-to-SQL task. The results show that SQL-specific fine-tuning consistently outperforms other model families, and that this gap becomes more pronounced at smaller model scales (around 1B parameters).

A key limitation observed in practice is the unstable behavior of a fixed model-prompt pair across different query categories, which makes single-pass inference brittle. To address this, we proposed three workflows and evaluated them against a baseline single-model/single-prompt setting. Overall, Prompt-Flow provides the best robustness under resource constraints, while Model-and-Prompt Flow achieves the strongest performance when accuracy is prioritized, at the cost of higher computation.

Future work should focus on learning or predicting an efficient ordering of model and prompt selection based on the input question, with the goal of reducing inference cost while maintaining high accuracy.

REFERENCES

- [1] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, and P. Liang, “On the opportunities and risks of foundation models,” 2022. [Online]. Available: <https://arxiv.org/abs/2108.07258>
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *Proceedings of NAACL-HLT*, 2019.
- [3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, and et al., “Language models are few-shot learners,” *Advances in Neural Information Processing Systems*, 2020, arXiv:2005.14165.
- [4] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, “Don’t stop pretraining: Adapt language models to domains and tasks,” 2020. [Online]. Available: <https://arxiv.org/abs/2004.10964>
- [5] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, “Finetuned language models are zero-shot learners,” in *International Conference on Learning Representations (ICLR)*, 2022.
- [6] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang et al., “Training language models to follow instructions with human feedback,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [8] S. Wu, O. Irsoy, S. Lu, V. Dabrovolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, “Bloomberggpt: A large language model for finance,” *arXiv preprint arXiv:2303.17564*, 2023.
- [9] R. Luo, L. Sun, Y. Xia, T. Qin, S. Zhang, H. Poon, and T.-Y. Liu, “Biogpt: Generative pre-trained transformer for biomedical text generation and mining,” *arXiv preprint arXiv:2210.10341*, 2022.
- [10] J. Kaplan, S. McCandlish, T. Henighan et al., “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.
- [11] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, “Q8bert: Quantized 8bit bert,” in *Proceedings of the 2019 EMNLP Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2019.
- [12] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, “Text-to-sql empowered by large language models: A benchmark evaluation,” 2023. [Online]. Available: <https://arxiv.org/abs/2308.15363>
- [13] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu et al., “Qwen2.5-coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.
- [14] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, “Deepseek-coder: When the large language model meets programming – the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [15] B. Rozière et al., “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [16] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, and H. Chen, “Codes: Towards building open-source language models for text-to-sql,” *arXiv preprint arXiv:2402.16347*, 2024.
- [17] M. Pourreza and D. Rafiei, “Din-sql: Decomposed in-context learning of text-to-sql with self-correction,” 2023. [Online]. Available: <https://arxiv.org/abs/2304.11015>
- [18] L. Nan, Y. Zhao, W. Zou, N. Ri, J. Tae, E. Zhang, A. Cohan, and D. Radev, “Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies,” 2023. [Online]. Available: <https://arxiv.org/abs/2305.12586>
- [19] A. Liu, X. Hu, L. Wen, and P. S. Yu, “A comprehensive evaluation of chatgpt’s zero-shot text-to-sql capability,” 2023. [Online]. Available: <https://arxiv.org/abs/2303.13547>
- [20] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, “Stanford alpaca: An instruction-following LLaMA model,” https://github.com/tatsu-lab/stanford_alpaca, 2023, gitHub repository.
- [21] S. Chang and E. Fosler-Lussier, “How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings,” 2023. [Online]. Available: <https://arxiv.org/abs/2305.11853>
- [22] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task,” 2019. [Online]. Available: <https://arxiv.org/abs/1809.08887>
- [23] A. Grattafiori et al., “The llama 3 herd of models,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.21783>
- [24] Qwen, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang et al., “Qwen2.5 technical report,” 2024. [Online]. Available: <https://arxiv.org/abs/2412.15115>
- [25] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier et al., “Mistral 7b,” 2023. [Online]. Available: <https://arxiv.org/abs/2310.06825>
- [26] DeepSeek-AI, X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du et al., “Deepseek llm: Scaling open-source language models with longtermism,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.02954>
- [27] P. Zhang, G. Zeng, T. Wang, and W. Lu, “Tinyllama: An open-source small language model,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.02385>
- [28] A. Lozhkov, R. Li, L. Ben Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei et al., “StarCoder 2 and the stack v2: The next generation,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.19173>
- [29] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, “Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality,” <https://lmsys.org/blog/2023-03-30-vicuna/>, 2023, accessed 2026-01-15.
- [30] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., “Llama 2: Open foundation and fine-tuned chat models,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.09288>
- [31] Defog.ai, “defog/sqlcoder-7b-2,” Hugging Face model card, 2024, accessed 2025-12-12. [Online]. Available: <https://huggingface.co/defog/sqlcoder-7b-2>
- [32] P. AI, “prem-research/prem-1b-sql,” Hugging Face model card, 2024, accessed 2025-12-12. [Online]. Available: <https://huggingface.co/prem-research/prem-1B-SQL>