

PV-SQL: Synergizing Database Probing and Rule-based Verification for Text-to-SQL Agents

Yuan Tian
Purdue University
West Lafayette, IN, USA

Tianyi Zhang
Purdue University
West Lafayette, IN, USA

Abstract

Text-to-SQL systems often struggle with deep contextual understanding, especially for complex queries with many subtle requirements. We present PV-SQL, an agentic method that addresses text-to-SQL generation failure through two complementary components, Probe and Verify. On the one hand, PV-SQL iteratively generates probe queries to retrieve concrete records from the database, resolving ambiguities related to value formats, column semantics, and inter-table relationships, thereby enabling deeper contextual understanding. On the other hand, PV-SQL leverages a rule-based method to extract and construct a checklist of verifiable conditions, which serves as executable test cases for iterative SQL refinement. This verification loop effectively reduces missing constraints in the generated query. Experiments on the BIRD benchmarks show that PV-SQL outperforms the best text-to-SQL baseline by 6% in execution accuracy and 20.8% in valid efficiency score while consuming fewer tokens.

1 Introduction

Text-to-SQL has emerged as a critical capability for democratizing database access, enabling non-experts to query structured data using natural language (Yu et al., 2018; Li et al., 2024). Despite remarkable progress driven by large language models (LLMs), existing systems face persistent challenges: (1) **schema understanding**, comprehending table relationships and column semantics; (2) **value grounding**, mapping natural language terms to exact database values, where schema information alone may not help; and (3) **constraint satisfaction**, ensuring the generated SQL faithfully captures all semantics expressed in the question.

Consider the question (Figure 1): “*What California orders customers were shipped late in 2023?*” A text-to-SQL system must resolve multiple ambiguities: Is “California” stored as the full name or abbreviated as “CA”? How is “late” represented?

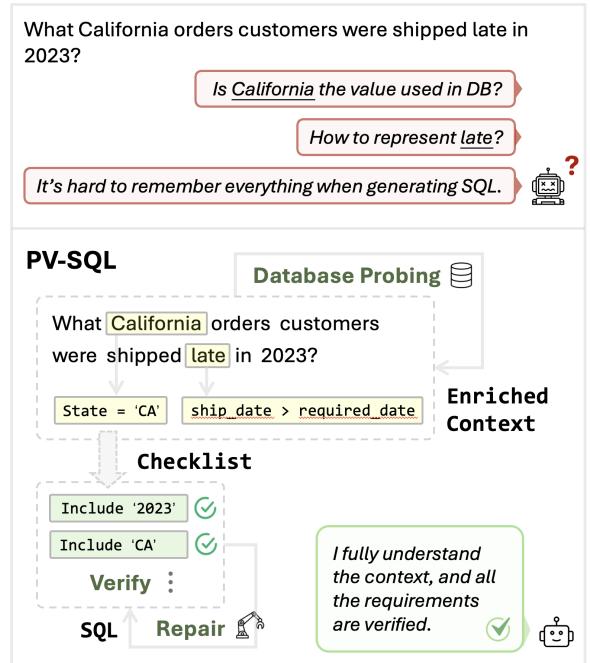


Figure 1: An example of how PV-SQL effectively solves a text-to-SQL task.

Does the database have a “late” flag? Answering these questions requires examining actual database values, yet existing methods typically rely only on schema descriptions, which do not contain such information.¹ Our empirical study (Section 3.2) reveals that approximately 41% of failed tasks stem from a misunderstanding of the database.

Previous work mainly uses static methods to enrich the context based on schema linking or semantic enrichment (Li et al., 2023; Ren et al., 2024; Caferoglu and Ulusoy, 2024; Lee et al., 2025). However, they only rely on pre-defined heuristics and do not adaptively explore database based on the question’s needs. Furthermore, even with a complete understanding of the database context, SQL

¹For example, Data Definition Language (DDL) defines the properties of a database by specifying data types, primary keys, and foreign keys, but does not mention database values.

generation inherently lacks a verification mechanism, such as test cases, to ensure that the generated SQL correctly meets all requirements. Models can easily generate syntactically correct SQL queries that silently return wrong answers, especially for complex queries. Although existing work leverages strategies such as refinement, candidate selection, or LLM-based verifications (Askari et al., 2024; Cen et al., 2024; Xu et al., 2025; Ni et al., 2023; Gong et al., 2025), a lack of reliable verification mechanisms remains.

To address these challenges, we propose PV-SQL, an agentic text-to-SQL method with two complementary components (Figure 1): (1) **Database Probing**: Rather than generating SQL based on schema alone, PV-SQL can generate and execute probe SQL queries to discover insights from database content, such as value formats, column semantics, and data distributions. (2) **Verify and Repair**: PV-SQL automatically extracts rule-based verifiable constraints from the question using pattern matching (e.g., requiring `DISTINCT` for “unique”, `LIMIT k` for “top-k”), and iteratively repairs the SQL until all constraints are satisfied.

These two components address complementary failure types. As shown in Figure 1, *Probe* discovers that “California” is stored as “CA” and “late” means `ship_date > required_date`. *Verify* then ensures that the generated SQL includes “CA” and “2023” as required. Essentially, probing enhances the *input* by enriching the context with concrete evidence, while verification enhances the *output* by ensuring desired semantic constraints are satisfied.

We evaluate PV-SQL by comparing it to seven strong baselines on three evaluation benchmarks across six base LLMs. PV-SQL achieves 65.12% execution accuracy and 86.9 valid efficiency score on BIRD, outperforming all baselines consistently across all benchmarks. Ablation studies confirm that both components contribute significantly.

2 Related Work

2.1 Text-to-SQL Methods

Large language models (LLMs) have transformed text-to-SQL from specialized semantic parsing into a general context understanding and reasoning challenge. DIN-SQL (Pourreza and Rafiei, 2023) decomposes the task into sub-problems including schema linking and query classification, while TA-SQL (Qu et al., 2024) incorporates task alignment to reduce hallucinations during schema linking and

logical synthesis. Some prompt-based frameworks encourage logical checking or multi-path reasoning during synthesis (Talaei et al., 2024; Pourreza et al., 2024). Recent multi-agent frameworks such as MAC-SQL (Wang et al., 2024) employ specialized agents for decomposition, generation, and refinement. Despite these advances, Rahaman and Gursoy (2024) show that even latest LLMs struggle with deep semantic understanding.

In response to this challenge, PV-SQL does not rely on schema alone or free-form self-correction. It enhances LLMs’ semantic understanding by (1) actively probing database content to resolve any ambiguity, and (2) rule-based verification to ensure that complex queries do not miss any semantics.

2.2 Context Enrichment

A large body of work improves text-to-SQL performance by enriching the model input with additional context, including semantic enrichment, entity linking, schema linking, and information retrieval. Semantic enrichment methods rewrite or augment the NL query with schema hints or grounded descriptions to make latent requirements explicit. Entity linking and schema linking methods align mentions in the NL query with database entities to reduce ambiguity. For example, RESDSQL (Li et al., 2023) prioritizes relevant schema elements using ranking-enhanced encoders, while E-SQL (Caferoglu and Özgür Ulusoy, 2025) directly enriches questions with linked schema elements and candidate predicates. Retrieval-augmented approaches such as PURPLE (Ren et al., 2024) incorporate external demonstrations, but rely on static retrieval based on surface similarity. However, these methods mainly focus on identifying relevant database entities (e.g, tables and columns), but do not effectively address the value grounding issue by mapping NL terms to relevant database values.

In contrast, PV-SQL employs adaptive database probing where the agent iteratively generates temporary SQL queries to explore the database, enabling question-specific context enrichment that similarity-based methods cannot achieve.

2.3 Verification-Driven Refinement

Verification-driven methods aim to improve text-to-SQL performance by validating and refining generated queries using feedback signals.

Inspired by self-consistency (Wang et al., 2022), one line of work samples multiple SQL candidates and selects the final output based on comparison

or majority voting (Li and Xie, 2024; Liu et al., 2025b; Chaturvedi et al., 2025). However, these methods are computationally expensive and unreliable, as it requires to generate many SQL candidates that may contain similar errors.

Another line of research conducts verification using external feedback signal, such as using the SQL execution results or additional models (Madaan et al., 2024; Askari et al., 2024; Cen et al., 2024; Gong et al., 2025). Self-Refine (Madaan et al., 2024) introduced the idea of using LLMs to critique their own outputs, but Huang et al. (2024) show that LLMs struggle to self-correct without external feedback. In code generation, test-driven methods such as CodeT (Chen et al., 2022) and Self-Debug (Chen et al., 2023) leverage generated test cases to refine generation, showing that explicit tests provide stronger verification than LLM self-verification. However, SQL queries inherently lack explicit test cases. LEVER (Ni et al., 2023) checks whether a generated SQL query is runnable but does not consider semantic correctness. TS-SQL (Xu et al., 2025) asks the LLM to synthesize test cases based on a translated Python version of the SQL query and then uses test results to refine the SQL. However, the test cases are still generated by LLMs, which is prone to errors and can lead to error accumulation. Furthermore, these methods often introduce significant computational overhead.

In contrast, PV-SQL uses a rule-based verifier that is reliable and lightweight. It leverages pattern-matching to extract verifiable constraints from the question, deterministically checks whether the SQL satisfies them, and iteratively repair the SQL to address violations. PV-SQL outperforms TS-SQL by 7.8% execution accuracy on BIRD (Section 6).

3 Problem Statement

3.1 Task Definition

Given a natural language question Q , a database D with schema \mathcal{S} , and optional evidence E (e.g., additional knowledge of the database), the text-to-SQL task is to generate a SQL query S such that executing S on D returns the correct answer to Q :

$$f : (Q, E, D, \mathcal{S}) \rightarrow S \quad (1)$$

We decompose the generation process into two stages: (1) *understanding* u , which builds an internal representation of the question semantics and database content, and (2) *synthesis* g , which trans-

lates this understanding into SQL. Formally:

$$u : (Q, E, D, \mathcal{S}) \rightarrow R, \quad g : R \rightarrow S \quad (2)$$

where R represents the model’s internal reasoning. Errors arise when either stage fails:

- **Database Misinterpretation** (\mathcal{E}_D): The understanding u produces incorrect assumptions about database content, e.g., value formats.
- **Question Misinterpretation** (\mathcal{E}_Q): The understanding u misses or misinterprets semantic constraints expressed in Q .
- **Synthesis Failure** (\mathcal{E}_S): The synthesis g fails to translate a correct understanding R into SQL.

3.2 Empirical Study on Error Distribution

Recent works (Ding et al., 2025; Liu et al., 2025a) highlight that ambiguity and context understanding are significant issues in text-to-SQL generation. Shen et al. (2025) shows that at least 30% of errors derive from misunderstanding the database schema or the natural language question. To understand reasons behind LLM-generated text-to-SQL errors, we analyze failed tasks from six LLMs (as discussed in Section 5.4) on the BIRD benchmark (Li et al., 2024). Following previous works (Zheng et al., 2023; Chirkova et al., 2025), we use GPT-4o as a judge to classify the error type. We manually measured the LLM judge’s correctness in 100 tasks, and the LLM judge achieved 88% accuracy. More details are discussed in Appendix D.

As shown in Table 1, database misinterpretation (\mathcal{E}_D) accounts for 41.3%, while question misinterpretation (\mathcal{E}_Q) accounts for 24.8%. This suggests that context misunderstanding significantly contributes to generation errors. This motivates our design: **Probe** addresses understanding errors ($\mathcal{E}_D + \mathcal{E}_Q$) by grounding the model in actual database content, which directly resolves database misinterpretation and mitigates question misinterpretation that stems from incomplete context. **Verify** addresses synthesis errors (\mathcal{E}_S) by ensuring no semantic constraints from the question are missed.

4 Method

Figure 2 illustrates PV-SQL. Given a question Q , evidence E , and database D with schema \mathcal{S} , our goal is to generate SQL query S that correctly answers Q . As motivated in Section 3, we address understanding errors ($\mathcal{E}_D + \mathcal{E}_Q$) by **Probe** (Figure 2, left) and synthesis errors (\mathcal{E}_S) by **Verify** (Figure 2, right). Algorithm 1 presents the procedure.

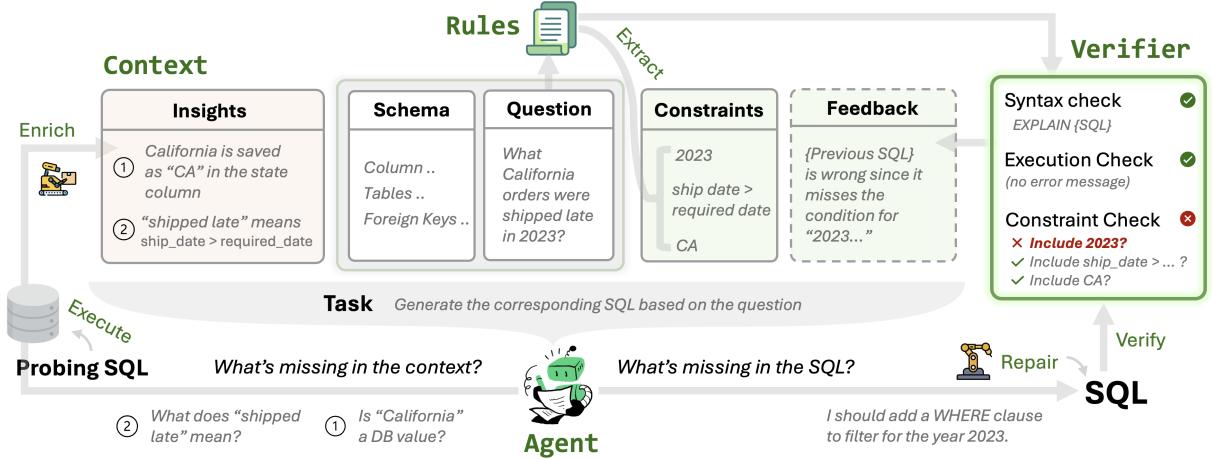


Figure 2: Overview of PV-SQL. **Left:** The agent generates probing SQL to discover database content and enriches the context with insights. **Top:** Rule-based extraction identifies semantic constraints from the question. **Right:** A rule-based verifier provides feedback for iterative repair until all checks pass.

Error Type	
Database Misinterpretation (\mathcal{E}_D)	41.3%
Question Misinterpretation (\mathcal{E}_Q)	24.8%
Synthesis Errors (\mathcal{E}_S)	33.9%

Table 1: Text-to-SQL Error distribution on BIRD.

4.1 Database Probing

The probing component iteratively queries the database to discover value formats and column semantics that the schema alone cannot reveal. Rather than including all database values in the prompt, which is infeasible for large databases and lacks question-specificity, probing enables uncertainty-driven exploration.

Probe Generation. At each iteration t , the LLM receives question Q , evidence E , schema overview O , and probe history $H_{t-1} = \{(p_1, r_1), \dots, (p_{t-1}, r_{t-1})\}$. The agent decides whether to issue another probe or proceed to SQL generation. If probing, it generates a SELECT query with a LIMIT clause to retrieve a bounded sample of relevant records.

Context Accumulation. After each probe execution, the agent interprets the results and summarizes what it has learned, identifying which columns are relevant to the question and extracting value mappings (e.g., “California” \rightarrow “CA”). These insights are accumulated into grounding context G , which enriches the subsequent SQL generation with verified database knowledge.

4.2 Constraint Extraction

PV-SQL extracts verifiable constraints from the question using pattern-matching rules. This design offers two advantages: (1) *reliability*, as pattern matching is simple and deterministic; (2) *efficiency*, as no additional LLM calls are required. We prioritize precision over recall: missing some constraints is acceptable, but false positives cause unnecessary repairs and introduce errors. We validate this choice empirically in Section 6.3 and Section 6.4.

Formally, each rule $r_i \in \mathcal{R}$ maps question patterns to a constraint predicate c_i checkable against SQL. Table 2 summarizes the ten constraint types we support, with representative patterns and their corresponding SQL checks. Complete pattern-matching rules are provided in Appendix G.

4.3 Verification and Refinement

Given a synthesized SQL query, PV-SQL detects potential errors through a multi-step pipeline²:

1. **Syntax Check:** PV-SQL first parses the SQL and checks its syntax correctness without execution using the EXPLAIN command.³
2. **Execution Check:** PV-SQL then executes the SQL against the database to catch runtime errors (e.g., type mismatches, division by zero).
3. **Constraint Check:** Unlike syntax and execution errors that databases catch automatically, semantic constraint violations require explicit ver-

²This design enables early termination, where queries failing syntax validation skip database execution entirely, reducing unnecessary computation.

³<https://www.geeksforgeeks.org/sql/explain-in-sql/>

Algorithm 1 PV-SQL

Require: Question Q , Evidence E , Database D ,
Max probes K , Max repairs M

Ensure: SQL query S

```
1: // Database Probing
2:  $H \leftarrow \emptyset$  {Probe history}
3:  $G \leftarrow \emptyset$  {Grounding context}
4: for  $t = 1$  to  $K$  do
5:   response  $\leftarrow \text{LLM}(Q, E, O, H)$ 
6:   if response.action = “done” then
7:     break
8:   end if
9:    $p_t \leftarrow \text{response.probe\_sql}$ 
10:   $r_t \leftarrow \text{Execute}(D, p_t)$ 
11:   $H \leftarrow H \cup \{(p_t, r_t)\}$ 
12:  Update  $G$  with insights from  $r_t$ 
13: end for
14: // Constraint Extraction
15:  $C \leftarrow \text{ExtractConstraints}(Q, E)$ 
16: // Verifiable Generation & Repair
17:  $S \leftarrow \text{LLM}(Q, E, G, O, C)$ 
18: for  $m = 1$  to  $M$  do
19:    $V \leftarrow \text{Violations}(S, C) \cup$ 
     ExecutionErrors( $S$ )
20:   if  $V = \emptyset$  then
21:     break
22:   end if
23:    $S \leftarrow \text{LLM}(Q, E, G, O, C, S, V)$ 
24: end for
25: return  $S$ 
```

ification. For each extracted constraint $c_i \in C$, we verify the presence of the corresponding SQL construct as specified in Table 2. Complete verification rules are provided in Appendix H.

Then, PV-SQL repair constraint-violating SQL by providing the LLM with the failed query, detected violations, and original context (question, evidence, grounding context from probing, and schema). Each violation leads to a descriptive error message that guides targeted repair. The process terminates when no violations remain, or after a maximum of 5 iterations to prevent infinite loop.

5 Experimental Setup

5.1 Benchmarks

We evaluate on three widely-used benchmarks. First, we use the development set of **Spider** (Yu et al., 2018), which includes 1034 tasks. Second, we use the development set of a more challeng-

Type	Example NL Patterns	SQL Verification
Distinct	unique, distinct, different	DISTINCT, GROUP BY
Top-K	top/first/best N	ORDER BY + LIMIT
Ranking	rank, position, standing	RANK(), ROW_NUMBER()
Count	how many, number of	COUNT(*)
Percent	percentage, ratio, rate	division (/) in SELECT
Sum	total, sum, combined	SUM()
Average	average, mean, avg	AVG()
Extreme	max, min, largest, smallest	MAX/MIN or LIMIT 1
Temporal	latest, earliest, newest	ORDER BY date column
Compare	more/less than, at least	>, <, >=, <=

Table 2: Simplified constraint extraction and corresponding SQL verification rules.

ing benchmark, **BIRD** (Li et al., 2024), which includes 1534 tasks. Furthermore, NL questions in BIRD are accompanied by evidence or hints that provide domain knowledge. Finally, following recent works (Sharma et al., 2025; Chen et al., 2025), we also use **Mini-Dev**, a curated 500-example subset of BIRD, developed from community feedback to contain only unambiguous, high-quality queries while preserving the original distribution of difficulty levels and database domains.

5.2 Metrics

Execution Accuracy (EX) measures whether the predicted SQL produces the same result as the gold SQL when executed.

Valid Efficiency Score (VES) (Li et al., 2024) combines execution accuracy with query efficiency, penalizing slow queries. This is specific to BIRD and Mini-Dev.

Token Consumption. Following the insight from test-time scaling (Snell et al., 2024), more reasoning generally leads to better performance. However, token consumption impacts cost and latency in practice. We report input tokens, output tokens, and average task completion time per query to evaluate the performance-cost efficiency.

5.3 Baselines

We select seven open-sourced and training-free⁴ SOTA text-to-SQL methods. **DAIL-SQL** (Gao et al., 2024) uses example selection and organization so LLMs see useful question-SQL pairs. **DIN-SQL** (Pourreza and Rafiei, 2023) decomposes the Text-to-SQL task into smaller subproblems and feeds intermediate solutions back to the LLM, boosting reasoning and accuracy compared

⁴For a fair comparison.

to naive prompting. **MAC-SQL** (Wang et al., 2024) is a multi-agent collaborative framework where specialized agents (decomposer, selector, refiner) work together with tools to generate and refine SQL queries for more complex scenarios. **E-SQL** (Caferoğlu and Özgür Ulusoy, 2025) enhances Text-to-SQL by directly enriching the question with relevant schema elements and candidate predicates, improving schema linking and handling ambiguous queries. **TA-SQL** (Qu et al., 2024) incorporates a task alignment strategy to reduce hallucinations by leveraging experience from similar tasks during schema linking and logical synthesis. **XiYan-SQL** (Liu et al., 2025b) generates multiple diverse SQL candidates via a multi-generator ensemble and then selects the best one with a trained selection model **TS-SQL** (Xu et al., 2025) adopts a test-driven refinement method that synthesizes test data and Python scripts that replicate the expected SQL behaviors. These scripts are executed to validate the generated SQL and produce execution feedback, guiding the correction of SQL errors. Since TS-SQL is not publicly available, we replicate it using prompts reported in their paper.

5.4 Base LLMs

Our evaluation includes six base LLMs. For closed-source models, we use GPT-4o and GPT-4.1-mini. For open-source models, we use GPT-OSS-20B, Gemma3-4B, Qwen3-4B, and Qwen3-0.6B, covering a range of model sizes from 0.6B to 20B parameters. All models disable the thinking mode for fair comparison and use a temperature of 0 for reproducibility. In Section 6, we report results using GPT-4o as the default base model when cross-model evaluation is not the focus. Additional results are provided in Appendix B.

Method	BIRD		Mini-Dev		Spider
	EX	VES	EX	VES	EX
GPT-4o	52.22	56.99	49.40	54.57	71.08
+ DAIL-SQL	54.10	58.77	50.00	50.84	71.18
+ DIN-SQL	54.80	56.75	51.00	54.79	69.05
+ MAC-SQL	58.70	62.77	57.80	64.65	72.73
+ E-SQL	59.10	64.65	57.40	64.23	75.63
+ TA-SQL	60.43	62.99	58.40	64.05	74.66
+ XiYan-SQL	57.60	63.96	52.20	56.31	72.73
+ TS-SQL	57.37	63.43	55.00	59.14	74.18
+ PV-SQL	65.12	75.55	63.80	74.63	77.66

Table 3: Execution accuracy (EX) and Valid Efficiency Score (VES) of different methods across benchmarks.

6 Results

6.1 Main Results

Tables 4 and 3 present our main results. PV-SQL consistently and significantly outperforms all baselines across different LLMs and benchmarks.

Cross-Benchmark Comparison As shown in Table 3, PV-SQL consistently outperforms all baselines across the three benchmarks, achieving 65.12% EX and 75.55% VES on BIRD, 63.80% EX and 74.63% VES on Mini-Dev, and 77.66% EX on Spider. The results indicate that PV-SQL generalizes well across different tasks.

Cross-Model Comparison As shown in Table 4, on BIRD, PV-SQL achieves 65.12% EX and 75.55% VES with GPT-4o, substantially outperforming all baselines by at least 4.69 and 12.56 absolute points, respectively. With GPT-4.1-mini, PV-SQL achieves 63.62% EX and 86.9% VES, surpassing all baselines by at least 3.42 and 20.79 absolute points. Notably, PV-SQL with GPT-4.1-mini achieves a remarkably high VES of 86.9%.

On open-weight models, PV-SQL also demonstrates strong generalizability, achieving the best performance across GPT-OSS-20B (59.45% EX, 61.99% VES), Qwen3-4B (49.61% EX, 51.47% VES), and Gemma3-4B (39.50% EX, 42.39% VES). However, we find that agentic methods do not work well on weaker models, as they have limited ability to process complex agentic workflows (Shen et al., 2024). For models smaller than 4B, most baselines reduce performance compared to simple chain-of-thought prompting (ZS-CoT), and Qwen3-0.6B is the only model where PV-SQL does not achieve the best performance.

6.2 Evaluation by Task Difficulties

Table 6 breaks down performance by task difficulty on BIRD. PV-SQL achieves consistent improvements across all difficulty levels.

PV-SQL outperforms the best baseline at every difficulty level: 71.03% vs. 67.57% (TA-SQL) on Simple, 56.68% vs. 52.16% (TA-SQL) on Moderate, and 54.48% vs. 45.52% (E-SQL) on Challenging queries. The improvement is most pronounced on Challenging queries (+8.96% EX over E-SQL), demonstrating that our probe-and-verify approach is particularly effective for complex queries where understanding data formats and verifying semantic constraints are critical. We observe similar patterns on Mini-Dev (see Appendix B).

Method	GPT-4o		GPT-4.1-mini		GPT-OSS-20B		Gemma3-4B		Qwen3-4B		Qwen3-0.6B	
	EX	VES	EX	VES	EX	VES	EX	VES	EX	VES	EX	VES
ZS-CoT	52.22	56.99	50.26	53.75	50.85	52.84	34.55	35.65	36.44	39.01	13.10	15.52
DAIL-SQL	54.10	58.77	50.40	55.05	49.39	48.22	27.32	27.41	28.81	29.99	8.15	8.50
DIN-SQL	54.80	56.75	53.80	58.22	50.03	46.56	22.23	20.57	23.44	22.51	6.84	7.13
MAC-SQL	58.70	62.77	55.60	60.62	53.59	51.50	37.70	37.36	39.76	40.88	11.60	12.95
E-SQL	59.10	64.65	54.80	60.95	53.95	53.05	26.20	29.14	41.66	46.38	12.84	13.06
TA-SQL	60.43	62.99	60.20	66.11	55.74	55.51	23.01	22.71	30.20	32.48	3.00	3.16
XiYan-SQL	57.60	63.96	51.60	58.99	35.98	38.03	20.79	30.69	36.40	38.31	6.60	6.93
TS-SQL	57.37	63.43	53.52	58.81	50.98	55.20	25.02	25.29	27.40	31.47	5.60	8.48
PV-SQL	65.12	75.55	63.62	86.9	59.45	61.99	39.50	42.39	49.61	51.47	11.08	15.39

Table 4: Execution accuracy (EX) and Valid Efficiency Score (VES) on the BIRD benchmark across different LLMs and text-to-SQL methods. Underline represents the best performance among all conditions.

Method	BIRD		Mini-Dev		Spider
	EX	VES	EX	VES	EX
PV-SQL	65.12	75.55	63.80	74.63	77.66
w/o Probe	62.13	70.07	60.60	69.78	73.79
w/o Repair	61.80	74.87	60.80	71.07	76.98
LLM verify	59.13	65.43	53.80	58.27	76.89

Table 5: Ablation Study of PV-SQL.

Method	Simple		Moderate		Challenging	
	EX	VES	EX	VES	EX	VES
GPT-4o	59.35	63.34	42.89	50.56	36.55	37.07
+ DAIL-SQL	63.14	70.07	42.03	43.41	35.17	35.88
+ DIN-SQL	60.97	63.30	46.12	48.57	42.76	41.09
+ MAC-SQL	65.62	71.05	49.78	53.13	43.45	40.77
+ E-SQL	65.08	70.31	51.29	54.67	45.52	60.54
+ TA-SQL	67.57	70.38	52.16	55.05	41.38	41.22
+ XiYan-SQL	64.43	71.30	48.28	54.85	44.14	46.25
+ TS-SQL	63.24	63.43	50.22	52.95	42.76	44.71
+ PV-SQL	71.03	81.42	56.68	65.37	54.48	63.51

Table 6: Execution accuracy (EX) and Valid Efficiency Score (VES) by task difficulty on BIRD.

6.3 Ablation Study

We conduct ablation studies to understand the contribution of each component in PV-SQL. Table 5 shows results when disabling the Probe (“w/o Probe”) or Repair (“w/o Repair”) components, and when replacing our rule-based verification with a LLM-based verification (“LLM Verify”).

Effect of Probing. Removing database probing leads to consistent performance degradation across all benchmarks, with execution accuracy (EX) dropping by 3.0 points on BIRD, 3.2 points on Mini-Dev, and 3.9 points on Spider. VES also

Component Evaluation Metrics		
Constraint Extraction Acc	99.39%	
Repair Success Rate	90.82%	
Repair Regression Rate	8.71%	

Table 7: Component evaluation of PV-SQL verification and repair. *Constraint Extraction Acc*: gold SQL pass rate. *Repair Success/Regression Rate*: violations resolved / constraints broken during repair.

drops substantially by 5.5 points on BIRD, 4.9 points on Mini-Dev, and 3.9 points on Spider. These results confirm that database probing helps the model understand data formats and value.

Effect of Repair. The verify-and-repair component is also important. Removing it reduces EX accuracy by 3.3% on BIRD, 3.0% on Mini-Dev, and 0.7% on Spider, indicating that the iterative refinement based on constraint violations is effective for correcting semantic errors.

Rule-based vs. LLM-based Verification. We compare our rule-based verification against an LLM-based alternative, where we follow previous works (Wang et al., 2023) to design the verification prompts (Appendix F). As shown in Table 8, rule-based verification achieves 6% higher execution accuracy while being 2× faster and using 45% fewer tokens, validating our design choice.

6.4 Component Evaluation

To better understand the performance of individual components, we conduct detailed component evaluation on BIRD. Table 7 summarizes the results.

Constraint Extraction. We evaluate constraint extraction accuracy by testing whether the gold

Method	BIRD				Mini-Dev			
	EX	In	Out	Time	EX	In	Out	Time
ZS-CoT	52.2	787	191	1.94	49.4	1927	211	2.1
DAIL-SQL	54.1	2446	50	0.8	50.0	2247	60	0.9
DIN-SQL	54.8	26629	629	25.0	51.0	26630	694	25.0
MAC-SQL	58.7	6067	387	4.7	57.8	6203	380	4.7
E-SQL	59.1	28177	642	21.4	57.4	29986	708	24.4
TA-SQL	60.43	6305	242	5.3	58.4	6365	240	6.7
XiYan-SQL	57.6	1689	128	2.4	52.2	1734	142	1.7
TS-SQL	57.4	1803	310	3.6	55.0	1891	362	4.1
PV-SQL	65.1	3805	248	3.4	63.8	4270	315	4.1
- w/o Probe	62.1	1068	67	1.0	60.6	1191	81	1.4
- w/o Repair	61.8	3521	223	3.1	60.8	4261	276	4.3
- LLM verify	59.1	4887	478	6.6	53.8	4983	508	6.7

Table 8: Efficiency Analysis across benchmarks and baselines. Execution accuracy (EX), input tokens, output tokens, and average task completion time (seconds).

SQL satisfies all extracted constraints. If all constraints are satisfied, the extraction is reliable; otherwise, the constraints may be irrelevant and potentially mislead the generation. Our rule-based method achieves 99.39% accuracy, suggesting that constraints used in PV-SQL are highly reliable.

Repair. We measure the repair performance by *Repair Success Rate* (violations successfully addressed) and *Regression Rate* (previously satisfied constraints broken). PV-SQL achieves a 90.82% success rate with only 8.71% regression rate, showing that the repairing reliably corrects errors without introducing new errors in most cases.

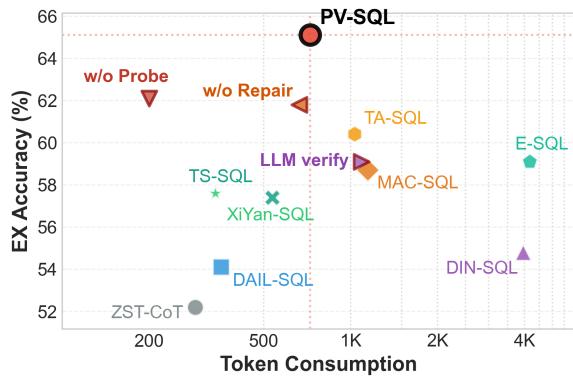


Figure 3: Accuracy vs. token consumption⁵ on BIRD.

6.5 Efficiency Analysis

Table 8 reports detailed efficiency metrics (token consumption, latency), while Figure 3 visu-

⁵Token consumption is computed as $\frac{1}{8} \times \text{input} + \text{output}$ tokens, reflecting API pricing where input tokens cost $\sim 8 \times$ less than output (<https://openai.com/api/pricing/>).

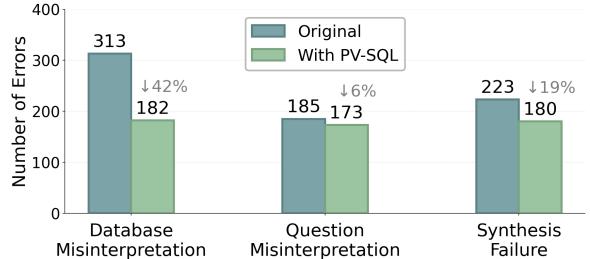


Figure 4: Error distribution before and after using PV-SQL on BIRD.

alizes the accuracy-efficiency landscape. PV-SQL achieves the best accuracy (65.1% on BIRD, 63.8% on Mini-Dev) with moderate token consumption. In contrast, DIN-SQL and E-SQL consume 7–8× more tokens and take 6–7× longer, yet achieve lower accuracy. The efficiency stems from our rule-based verification, which avoids expensive LLM calls. Replacing it with “LLM verify” increases tokens significantly while decreasing accuracy by 6%. Notably, removing Probe achieves the best efficiency with only a 3% drop in accuracy.

6.6 Error Analysis

To understand how PV-SQL reduces errors, we apply the same error classification method in Section 3, on the BIRD benchmark. Figure 4 shows the error distribution before and after applying PV-SQL. Specifically, there is a 42% reduction in database misinterpretation errors, validating our hypothesis that probing helps ground the model in actual database content. Synthesis failures also decrease by 19%, demonstrating that the verify-and-repair loop effectively catches semantic constraint violations. Question misinterpretation shows a modest 6% reduction, which is expected since this error type often stems from inherent ambiguity in natural language rather than missing context. To demonstrate how PV-SQL addresses errors, we include a case study in Appendix A.

7 Conclusion

We presented PV-SQL, an agentic method for robust text-to-SQL generation. By probing the database to ground the input context and extracting explicit constraint to verify the output, PV-SQL addresses complementary sources of error. Experiments demonstrate that probing and verification can significantly and consistently enhance text-to-SQL performance, with less token consumptions.

Limitations

Our constraint extraction rules were developed through empirical observation of common text-to-SQL patterns. While the current rule set covers frequently occurring constraints (e.g., aggregation, sorting, filtering), it does not exhaustively capture all possible semantic requirements. Extending the rule set to handle more nuanced or domain-specific constraints remains an avenue for future work.

Additionally, our empirical study about text-to-SQL error analysis rely on LLM-as-a-judge. While our human annotation shows it is reliable and it provides scalable evaluation, it introduces potential inaccuracies. The automated assessments should be interpreted as estimates rather than ground truth.

References

Arian Askari, Christian Poelitz, and Xinye Tang. 2024. MAGIC: Generating self-correction guidelines for in-context text-to-SQL. *arXiv preprint arXiv:2406.12692*.

Hasan Alp Caferoglu and Özgür Ulusoy. 2024. E-SQL: Direct schema linking via question enrichment in text-to-SQL. *arXiv preprint arXiv:2409.16751*.

Hasan Alp Caferoglu and Özgür Ulusoy. 2025. E-sql: Direct schema linking via question enrichment in text-to-sql. *Preprint, arXiv:2409.16751*.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2024. SQLFixAgent: Towards semantic-accurate SQL generation via multi-agent collaboration. *arXiv preprint arXiv:2406.13408*.

Saumya Chaturvedi, Aman Chadha, and Laurent Bind-schaedler. 2025. Sql-of-thought: Multi-agentic text-to-sql with guided error correction. *Preprint, arXiv:2509.00581*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *Preprint, arXiv:2207.10397*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *Preprint, arXiv:2304.05128*.

Zui Chen, Han Li, Xinhao Zhang, Xiaoyu Chen, Chun-yin Dong, Yifeng Wang, Xin Cai, Su Zhang, Ziqi Li, Chi Ding, Jinxu Li, Shuai Wang, Dousheng Zhao, Sanhai Gao, and Guangyi Liu. 2025. Rubiksql: Life-long learning agentic knowledge base as an industrial nl2sql system. *Preprint, arXiv:2508.17590*.

Nadezhda Chirkova, Tunde Oluwaseyi Ajayi, Seth Aycock, Zain Muhammad Mujahid, Vladana Perlić, Ekaterina Borisova, and Markarit Vartampetian. 2025. Llm-as-a-qualitative-judge: automating error analysis in natural language generation. *Preprint, arXiv:2506.09147*.

Zhongjun Ding, Yin Lin, and Tianjing Zeng. 2025. Ambisql: Interactive ambiguity detection and resolution for text-to-sql. *Preprint, arXiv:2508.15276*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. DAIL-SQL: Optimized llm prompt for text-to-SQL. In *Proceedings of the VLDB Endowment*, volume 17, pages 950–961.

Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. Sqleens: An end-to-end framework for error detection and correction in text-to-sql. *Preprint, arXiv:2506.04494*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. *Preprint, arXiv:2310.01798*.

Jihyung Lee, Jin-Seop Lee, Jaehoon Lee, YunSeok Choi, and Jee-Hyong Lee. 2025. DCG-SQL: Enhancing in-context learning for text-to-SQL with deep contextual schema link graph. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15397–15412, Vienna, Austria. Association for Computational Linguistics.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence, AAAI’23/IAAI’23/EAAI’23*. AAAI Press.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Li, Bei Hui, and Yongbin Li. 2024. Can LLM already serve as a database interface? a Big bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36.

Zhenwen Li and Tao Xie. 2024. Using LLM to select the right SQL query from candidates. *arXiv preprint arXiv:2401.02115*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025a. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *Preprint, arXiv:2408.05109*.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li,

Bolin Ding, and Jingren Zhou. 2025b. *Xiyan-sql: A novel multi-generator framework for text-to-sql*. *Preprint*, arXiv:2507.04701.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *Preprint*, arXiv:2410.01943.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Preprint*, arXiv:2304.11015.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. *Preprint*, arXiv:2405.15307.

Md Mahadi Hassan Rahaman and Mehmet Emre Gursoy. 2024. Evaluating sql understanding in large language models. *Preprint*, arXiv:2410.10680.

Minseok Ren, Sijie Jin, Soobin Son, Yeonsu Kwon, Dong-Gun Lee, Hwi-Jun Yang, and Kyomin Jung. 2024. Purple: Making a large language model a better sql writer. *Preprint*, arXiv:2403.20014.

Chetan Sharma, Ramasuri Narayanan, Soumyabrata Pal, Kalidas Yeturu, Shiv Kumar Saini, and Koyel Mukherjee. 2025. TTD-SQL: Tree-guided token decoding for efficient and schema-aware SQL generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1287–1298, Suzhou (China). Association for Computational Linguistics.

Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazheng Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025. A study of in-context-learning-based text-to-sql errors. *Preprint*, arXiv:2501.09310.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. Small LLMs are weak tool learners: A multi-LLM agent. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16658–16680, Miami, Florida, USA. Association for Computational Linguistics.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *Preprint*, arXiv:2408.03314.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Wenbo Xu, Haifeng Zhu, Liang Yan, Chuanyi Liu, Peiyi Han, Shaoming Duan, and Jeff Z. Pan. 2025. TS-SQL: Test-driven self-refinement for text-to-SQL. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 2864–2889.

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. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zuhuan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

A Case Study

Question: List the top 3 unique product categories by percentage of orders from California customers that were shipped late in 2023.

[Probe]

Probe 1:

```
SELECT DISTINCT state FROM customers LIMIT 5
Result: [('CA',), ('NY',), ('TX',), ...]
```

Learned:

State stored as 2-letter codes

Probe 2:

```
SELECT ship_date, required_date FROM orders
LIMIT 3
Result: [(2023-01-15', '2023-01-10'), ...]
```

Learned:

Dates as YYYY-MM-DD

“Late” means `ship_date > required_date`

[Generate]

```
SELECT p.category, CAST(SUM(CASE WHEN
o.ship_date > o.required_date THEN 1 ELSE
0 END) AS REAL) * 100 / COUNT(*) FROM orders o
JOIN customers c ON o.customer_id = c.id JOIN
products p ON o.product_id = p.id WHERE c.state
= 'CA' GROUP BY p.category
```

[Verify]

Extracted Constraints:

- ✓ Percentage calculation required
- ✓ DISTINCT or GROUP BY for uniqueness
- ✗ “top 3” requires ORDER BY ... LIMIT 3 → **Violation**
- ✗ “2023” must appear in SQL → **Violation**

[Repair]

```
SELECT p.category, CAST(SUM(...) AS REAL) * 100 / COUNT(*) AS late_pct FROM orders o
JOIN customers c ... JOIN products p ... WHERE
c.state = 'CA' AND strftime('%Y', o.order_date)
= '2023' GROUP BY p.category ORDER BY late_pct
DESC LIMIT 3
```

[Verify]

✓ All constraints satisfied

Figure 5: Case study showing how probing discovers value formats and verification catches multiple semantic errors (missing year filter, missing LIMIT).

Figure 5 demonstrates how probing and verification work together in PV-SQL. The question asks for the top 3 product categories by late shipment percentage for California customers in 2023. Through *Probe*, the agent discovers that states are stored as 2-letter codes and dates follow the YYYY-MM-DD format, revealing that “late” means `ship_date > required_date`. Without probing, these value formats would be difficult to infer from schema alone.

Verify extracts four constraints from the question: percentage calculation, uniqueness (via GROUP BY), top-3 limit, and year filter. The initial SQL satisfies two constraints but misses the ORDER BY

... LIMIT clause and the 2023 filter. *Repair* fixes both violations (highlighted in green), and a final verification confirms all constraints are satisfied. This iterative process reduces the burden on the agent to produce a perfect query in one pass.

Constraint	Trigger Patterns
Distinctness	“unique”, “distinct”, “different”, “no duplicate”, “deduplicate”
Top-K	“top N ”, “first N ”, “bottom N ”, “highest N ”, “lowest N ”, “best N ”, “worst N ”
Ranking	“rank”, “ranking”, “position”, “placed”, “standing”
Counting	“how many”, “count”, “number of”, “total number”, “quantity of”
Percentage	“percentage”, “percent”, “%”, “ratio”, “rate”, “proportion”, “fraction of”
Summation	“total”, “sum”, “overall”, “combined”, “aggregate”
Average	“average”, “mean”, “avg”, “on average”, “typical”
Extreme	“maximum”, “minimum”, “max”, “min”, “largest”, “smallest”, “most”, “least”, “highest”, “lowest”
Temporal	“latest”, “earliest”, “most recent”, “newest”, “oldest”, “last”, “first” (with time context)
Comparison	“more than”, “less than”, “greater than”, “fewer than”, “at least”, “at most”, “no more than”, “exceeds”

Table 9: Constraint extraction rules mapping question patterns to semantic constraints. Patterns are matched case-insensitively against the question text.

B Additional Experiment Results

This section contains additional experimental results that complement the main paper. We provide: (1) baseline LLM performance by difficulty across all models, (2) cross-benchmark comparison with GPT-4.1-mini to complement Table 3, (3) performance breakdown by difficulty on Mini-Dev with GPT-4o to complement Table 6, and (4) performance breakdown by difficulty for GPT-4.1-mini.

C Prompts Used in PV-SQL

We present the four prompts used in PV-SQL: Probe, Generation, Repair, and Error Analysis.

C.1 Probe Prompt

The Probe prompt guides the LLM to generate probing SQL queries that reveal database content,

Constraint	SQL Verification Criteria
Distinctness	Presence of <code>DISTINCT</code> keyword in <code>SELECT</code> , or <code>GROUP BY</code> clause that implies uniqueness
Top-K	Presence of <code>LIMIT N</code> clause where N matches the requested count; combined with <code>ORDER BY</code> for “top” queries
Ranking	Presence of window function: <code>RANK()</code> , <code>DENSE_RANK()</code> , or <code>ROW_NUMBER()</code> with <code>OVER</code> clause
Counting	Presence of <code>COUNT(*)</code> or <code>COUNT(column)</code> in <code>SELECT</code> clause
Percentage	Presence of division operator <code>(/)</code> or multiplication by <code>100.0</code> in <code>SELECT</code> ; often with <code>CAST</code> for float division
Summation	Presence of <code>SUM(column)</code> aggregation function in <code>SELECT</code> clause
Average	Presence of <code>AVG(column)</code> aggregation function in <code>SELECT</code> clause
Extreme	Presence of <code>MAX()</code> or <code>MIN()</code> function; or <code>ORDER BY</code> with <code>LIMIT 1</code> pattern
Temporal	<code>ORDER BY</code> clause on date/time column; direction (ASC/DESC) matches “earliest”/“latest”
Comparison	<code>WHERE</code> or <code>HAVING</code> clause with comparison operators <code>(>, <, >=, <=)</code> matching the constraint

Table 10: Constraint verification rules. Each rule checks for specific SQL constructs that satisfy the semantic requirement.

value formats, and data distributions not apparent from schema alone. Given the question, evidence, schema, and results from prior probes, it outputs a JSON object containing a probe SQL query and discovered value mappings. This grounds the text-to-SQL task by understanding actual database content before generating the final query.

C.2 Generation Prompt

The Generation prompt produces the initial SQL query using enriched context from the probing phase. It takes the question, evidence, schema, value mappings, probe observations, and extracted constraints as input, and outputs a single SQL query that leverages the grounded understanding of database content.

C.3 Repair Prompt

The Repair prompt fixes SQL queries that violate constraints detected by the rule-based verifier. Given the original context, the faulty SQL, and specific violation messages from the verifier, it outputs a corrected SQL query. This enables iterative

refinement by addressing specific constraint violations rather than regenerating from scratch.

D Error Analysis Prompt

The Error Analysis prompt is used for post-hoc analysis of failed predictions, classifying errors to understand model weaknesses. Given the question, evidence, schema, ground truth SQL, predicted SQL, and execution errors, it outputs a JSON object containing the error type, reasoning, and specific issue. This enables systematic categorization of errors for comparative analysis across methods.

E Probe Quality Evaluation Prompt

This section presents the prompt used for LLM-based evaluation of probe quality in Section 6. We use GPT-4o as the judge to assess each probe query along three dimensions: relevance to the question, whether it provides new insights beyond schema information, and redundancy with previous probes.

F LLM-based Verification Prompts

This section presents the prompts used in the “LLM verify” ablation variant described in Section 6.3. In this variant, we replace the rule-based constraint extraction and verification with LLM-based approaches. While this approach is more flexible, it achieves lower accuracy and higher cost compared to rule-based verification.

F.1 LLM-based Constraint Extraction

The LLM-based constraint extraction prompt asks the model to analyze the question and extract semantic constraints that the SQL query must satisfy. Unlike the rule-based approach (Section G), this relies on the LLM’s understanding to identify requirements.

F.2 LLM-based SQL Verification

The LLM-based verification prompt asks the model to verify whether a generated SQL query correctly satisfies all requirements from the original question. This replaces the rule-based verification that checks for specific SQL constructs.

G Constraint Extraction Rules

This section expands on the constraint extraction described in Section 4.2. Table 9 presents the complete set of pattern-matching rules for extracting semantic constraints from natural language questions. Each rule maps question patterns (keywords

or phrases) to a constraint type that can be verified against the generated SQL.

H Constraint Verification Rules

This section expands on the violation detection described in Section 4.3. Table 10 presents the verification rules that check whether a generated SQL query satisfies each extracted constraint. Verification is performed by parsing the SQL and checking for the presence of required constructs.

We implement verification using SQL parsing with the `sqlparse` library. For each constraint type, we traverse the parsed AST to check for the required tokens or clauses. The verifier returns a list of violated constraints with descriptive error messages (e.g., “Question asks for unique values but SQL lacks `DISTINCT` or `GROUP BY`”).

Method	BIRD		Mini-Dev		Spider	
	EX	VES	EX	VES	EX	
GPT-4.1-mini	50.26	53.75	49.40	54.86	56.99	
+ DAIL-SQL	53.80	62.71	50.40	55.05	58.77	
+ DIN-SQL	55.50	62.38	53.80	58.22	56.75	
+ MAC-SQL	58.30	63.97	55.60	60.62	62.77	
+ E-SQL	57.80	65.03	54.80	60.95	64.65	
+ TA-SQL	61.10	69.62	60.20	66.11	68.83	
+ XiYan-SQL	55.90	63.92	51.60	58.99	63.96	
+ TS-SQL	53.52	58.81	50.20	54.87	63.43	
+ PV-SQL	63.62	72.78	60.20	67.15	75.55	

Table 11: Cross-benchmark comparison with GPT-4.1-mini as the base model.

Method	Simple		Moderate		Challenging	
	EX	VES	EX	VES	EX	VES
GPT-4o	60.81	64.27	48.00	57.00	36.27	34.53
+ DAIL-SQL	66.89	68.13	47.60	48.24	31.37	32.12
+ DIN-SQL	64.86	65.04	48.40	55.42	37.25	38.37
+ MAC-SQL	72.30	82.81	55.20	62.60	43.14	43.34
+ E-SQL	69.59	84.82	55.60	58.94	44.12	47.32
+ TA-SQL	72.97	81.91	57.60	62.82	39.22	41.17
+ XiYan-SQL	65.54	67.30	50.00	56.58	38.24	39.72
+ TS-SQL	67.57	76.82	54.00	55.93	39.22	41.37
+ PV-SQL	76.35	87.28	62.80	76.21	48.04	52.40

Table 12: Performance by difficulty on Mini-Dev with GPT-4o.

Method	Simple		Moderate		Challenging	
	EX	VES	EX	VES	EX	VES
GPT-4.1-mini	58.05	61.68	39.44	44.29	35.17	33.42
+ DAIL-SQL	61.30	74.23	44.60	47.74	35.90	37.20
+ DIN-SQL	61.80	69.65	48.30	54.45	37.90	41.35
+ MAC-SQL	65.30	73.68	47.60	50.84	48.30	44.11
+ E-SQL	64.00	74.42	48.50	51.70	47.60	47.78
+ TA-SQL	68.10	78.22	51.90	57.59	45.50	53.22
+ XiYan-SQL	63.40	72.77	46.10	52.25	39.30	44.81
+ TS-SQL	60.32	68.66	45.26	45.77	36.55	37.69
+ PV-SQL	69.73	79.21	56.47	66.46	47.59	51.95

Table 13: Performance by difficulty on BIRD with GPT-4.1-mini.

Method	Simple		Moderate		Challenging	
	EX	VES	EX	VES	EX	VES
GPT-4.1-mini	63.51	73.28	48.40	53.28	31.37	31.98
+ DAIL-SQL	67.60	78.02	47.60	50.47	32.40	32.96
+ DIN-SQL	64.20	73.58	54.80	57.33	36.30	38.13
+ MAC-SQL	70.90	78.36	52.40	58.34	41.20	40.47
+ E-SQL	66.20	76.53	54.40	58.99	39.20	43.13
+ TA-SQL	76.40	86.07	58.40	64.63	41.20	40.77
+ XiYan-SQL	65.50	76.20	49.60	55.63	36.30	42.26
+ TS-SQL	62.84	74.13	48.80	50.61	35.29	37.35
+ PV-SQL	73.65	86.52	59.20	64.04	43.14	46.68

Table 14: Performance by difficulty on Mini-Dev with GPT-4.1-mini.

Difficulty	GPT-4o		GPT-4.1-mini		GPT-OSS-20B		Gemma3-4B		Qwen3-4B		Qwen3-0.6B	
	EX	VES	EX	VES	EX	VES	EX	VES	EX	VES	EX	VES
Simple	65.00	70.06	60.80	67.88	63.57	66.16	44.22	45.58	44.97	49.86	18.70	22.03
Moderate	45.30	48.74	42.70	46.18	45.04	47.35	21.12	21.88	23.49	22.74	4.96	6.14
Challenging	38.60	40.35	33.10	34.41	40.00	40.07	15.86	16.36	23.45	21.90	3.45	4.02
All	56.50	60.80	52.70	58.16	55.74	58.00	34.55	35.65	36.44	39.01	13.10	15.52

Table 15: ZS-CoT performance by difficulty level across different LLMs on BIRD.

Difficulty	GPT-4o		GPT-4.1-mini	
	EX	VES	EX	VES
Simple	60.81	64.27	66.90	79.65
Moderate	48.00	57.00	48.40	54.67
Challenging	36.27	34.53	31.40	32.99
All	49.40	54.57	50.40	57.64

Table 16: ZS-CoT performance by difficulty level on Mini-Dev..

Prompt 1: Probe

```

## Background
You are helping to solve a text-to-SQL problem.
Before writing the final SQL query, you can run
exploratory “probe” queries on the database to
understand its content. Probes help discover
addition knowledge that are not apparent from
the schema alone.

## Task
Analyze the question and schema, then decide
whether to request a probe query or proceed to
SQL generation.

## Context
Question: {question}
Evidence (hints provided with the question):
{evidence}
Database Schema: {schema}
Prior Probes (queries you already ran and their
results): {prior_probe_results}

## Instructions
- If you need more information, generate a
probe query (e.g., SELECT DISTINCT column FROM
table LIMIT 5).
- If you have enough information, set action to
“done”.
- Record any value mappings you discover (e.g.,
“California” maps to “CA” in the database).

## Output Format (JSON)
{
  "action": "probe" | "done",
  "probe_sql": "SELECT ...",
  "relevant_columns": {"table": ["col1",
  "col2"]},
  "value_mappings": {"term_in_question": "exact_db_value"}
}

```

Prompt 2: Generation

```

## Task
Write a SQL query to answer the question based
on the provided context.

## Context
Question: {question}
Evidence (hints provided with the question):
{evidence}
Database Schema: {schema}
Probe Observations (results from exploratory
queries run on the database, showing actual
values and formats):
{probe_results}

## Rules
- Write a single SQL statement only.
- Return only what is asked (no extra columns).
- Follow evidence/hints strictly when provided.
- Use exact database values discovered from
probes (e.g., use “CA” not “California” if
probes showed state codes).

## Output
SQL query only.

```

Prompt 3: Repair

```
## Background
A SQL query was generated but failed verification. The verifier checks for constraint violations (e.g., missing DISTINCT when the question asks for unique values, missing LIMIT for top-k queries) and execution errors (e.g., invalid column names, syntax errors).

## Task
Fix the SQL query to resolve the detected violations.

## Context
Question: {question}
Evidence (hints provided with the question): {evidence}

Database Schema: {schema}
Original SQL (the query that failed verification): {original_sql}

## Violations Detected (errors found by the verifier):
{violation_messages}

## Instructions
- Carefully address each violation listed above.
- Output the corrected SQL only (no explanation).
```

Prompt 4: Error Analysis

```
## Background
You are analyzing why a text-to-SQL model failed to generate the correct query. By comparing the predicted SQL with the ground truth, you will classify the root cause of the error to help understand model weaknesses.

## Task
Analyze the failure and classify it into one error category.

## Context
Question: {question}
Evidence (hints provided with the question): {evidence}
Database Schema: {schema}
Ground Truth SQL (the correct answer): {gold_sql}
Predicted SQL (what the model generated): {predicted_sql}
Execution Error (if the predicted SQL failed to run): {exec_error}

## Error Types
Classify into exactly one category:
1. DATABASE_MISINTERPRETATION:
Failed to understand database content, schema, or relationships.
Examples: wrong table/column, misunderstood foreign keys or data formats.

2. QUESTION_MISINTERPRETATION:
Misinterpreted the question.
Examples: wrong filter conditions, misunderstood aggregation requirements.

3. SQL_SYNTHESIS_FAILURE:
Understood context correctly but failed to generate correct SQL.
Examples: wrong JOIN syntax, missing clauses, syntax errors.

## Output Format (JSON)
{
  "error_type": "...",
  "reasoning": "...",
  "specific_issue": "..."
}
```

Prompt 5: Probe Quality Evaluation

```
## Background
You are evaluating the quality of probe queries
for text-to-SQL grounding. Probes are
exploratory SQL queries that help understand
database content before generating the final
query.

## Task
For each probe query, assess three aspects:
1. RELEVANCE: Is this probe relevant to
answering the question? (yes/no)
2. NEW_INSIGHT: Does this probe provide
information not available from schema alone?
(yes/no)
  - Schema-only info: table/column names, data
  types, foreign keys
  - New insights: actual data values, data
  formats, value distributions, NULL patterns

3. REDUNDANT: Does this probe duplicate
information from previous probes? (yes/no)

## Context
Question: {question}
Evidence: {evidence}
Schema: {schema}
Probes to evaluate: {probes}

## Output Format (JSON)
{
  "evaluations": [
    {"probe_index": 0, "relevant": true/false,
     "new_insight": true/false, "redundant":
     true/false,
     "reasoning": "..."},
    ...
  ]
}
```

Prompt 6: LLM-based Constraint Extraction

```
## Task
You are a SQL requirements analyst. Extract
semantic constraints from the natural language
question and evidence.

## Context
Question: {question}
Evidence: {evidence}

## Instructions
Focus on identifying:
- DISTINCT requirements (unique, different,
distinct values)
- Aggregation needs (count, sum, avg, max, min)
- Ordering/ranking requirements (top N,
highest, lowest, oldest, newest)
- Percentage/ratio calculations
- Comparison operators (greater than, less
than, equal to)
- Grouping requirements
- NULL handling needs

- Any specific value filtering mentioned

## Output Format (JSON)
{
  "constraints": [
    {
      "type": "distinct|limit|aggregation|...",
      "description": "human-readable
description",
      "sql_hint": "what SQL construct should
be used"
    }
  ],
  "output_requirements": {
    "expected_columns": ["col1", "col2"],
    "expected_type": "single_value|list|count|..."
  }
}
```

Prompt 7: LLM-based SQL Verification

```
## Task
You are a SQL verification expert. Verify if
the generated SQL query correctly satisfies
all requirements from the original question.

## Context
Question: {question}
Evidence: {evidence}
Extracted Constraints: {constraints}
Schema: {schema}
Generated SQL: {sql}

## Verification Checklist
Verify ALL aspects:
1. Structural validity: Is it a single valid
SELECT/WITH statement?
2. Semantic correctness: Does the SQL return
what the question asks?
3. Filter correctness: Are all
filters/conditions applied?
4. Aggregation correctness: Is aggregation
correct (COUNT vs COUNT(DISTINCT))?
5. Ordering correctness: Is ordering/limit
correct for "top N" queries?
6. JOIN correctness: Are JOINs correct and
complete?
7. Output format: Is the output format correct?

## Instructions
Be thorough but avoid false positives. Only
flag issues that are clearly problems.

## Output Format (JSON)
{
  "is_valid": true/false,
  "issues": [
    {
      "severity": "error|warning",
      "category": "syntax|semantic|missing_constraint|...",
      "description": "detailed description of
the issue",
      "suggestion": "how to fix it"
    }
  ]
}
```