ExpeSQL: An Efficient, Experience-Guided Decompositional Search Framework for Text-to-SQL

Zerui Yang
City University of Hong Kong
Hong Kong, China
zeruiyang2-c@my.cityu.edu.hk

Siyu Yan
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China

Boyan Li
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China

Yanwei Xu
Huawei Technologies Ltd.
Hong Kong, China

Linqi Song
City University of Hong Kong
Hong Kong, China

Yudai Matsuda
City University of Hong Kong
Hong Kong, China

Wei Han
Huawei Technologies Ltd.
Hong Kong, China

Bo Bai
Huawei Technologies Ltd.
Hong Kong, China

ABSTRACT

Large language models have advanced Text-to-SQL, yet enterprise deployment remains difficult due to complex, evolving schemas, domain shift, privacy constraints, and latency/cost budgets. We introduce **ExpeSQL**, a zero-shot, open-source–compatible, and efficient framework that combines divide-and-conquer reasoning, Best-of-N candidate selection, and self-critique with experience-guided refinement. ExpeSQL scopes relevant schema, decomposes questions, executes verifiable sub-SQLs, and aggregates candidates via result-based filtering and majority voting; failures trigger diagnostic backtracking that stores structured remedies in long-term memory to guide subsequent iterations without parameter updates. On BIRD-dev, ExpeSQL achieves 67.5% execution accuracy with open-source models, reducing token generation by up to 87% and inference latency by 96% relative to Alpha-SQL at comparable accuracy, thereby achieving a significantly better accuracy-efficiency trade-off. Our results establish a new paradigm for deployable, self-improving Text-to-SQL systems in dynamic, real-world environments.

VLDB Workshop Reference Format:
Zerui Yang, Siyu Yan, Boyan Li, Yanwei Xu, Linqi Song, Yudai Matsuda, Wei Han, and Bo Bai. ExpeSQL: An Efficient, Experience-Guided Decompositional Search Framework for Text-to-SQL. VLDB 2026 Workshop: ADS 2026: The Joint Workshop on Agentic Data Systems and Data-Centric AI (The 1st ADS & 3rd DATAI).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts will be made available at https://github.com/yaoge777/ExpeSQL following publication.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

Large language models (LLMs) have revolutionized Text-to-SQL systems [18, 48], enabling non-expert users to query databases through natural language and significantly lowering the barrier to data access. While promising, the practical deployment of LLM-based Text-to-SQL systems in enterprise environments remains challenging [5]. Real-world databases are often large, schema-complex, and subject to frequent updates and domain shifts—conditions under which standard LLMs exhibit poor robustness, generating syntactically valid but semantically incorrect queries due to hallucinated joins, misaligned columns, or ambiguous user intents. Moreover, enterprise applications demand strict data privacy, low-latency responses, and minimal operational overhead, all of which are difficult to achieve with current approaches. To improve accuracy, existing methods typically rely on either training (TR) (e.g., supervised fine-tuning or reinforcement learning) [19] or test-time scaling (TTS) [14, 43].

TR methods, such as Reasoning-SQL [26], leverage domain-specific data and advanced training objectives (e.g., GRPO) to achieve high performance—up to 72.29 execution accuracy on the BIRD dev set [16] with a 14B open-source model. However, TR-based systems face critical barriers to enterprise adoption: they require extensive labeled training data, incur high retraining costs when schemas evolve, and often fail to generalize to unseen domains or new databases. More critically, fine-tuning typically involves sending sensitive SQL and schema information to third-party services or internal training pipelines, raising serious data leakage risks in regulated environments.

TTS Methods. To address these limitations, TTS has recently emerged as a compelling alternative to fine-tuning [9]. For instance, CHASE-SQL [23] leverages three strategies—divide-and-conquer, query planning, and online synthetic example generation—to produce multiple candidate SQL queries, which are then selected by a fine-tuned agent. However, like many TTS approaches, it depends on closed-source APIs, incurring high operational costs and raising privacy concerns in enterprise settings.

Alpha-SQL [14] stands out as a notable exception: a purely test-time scaling method built entirely on the open-source Qwen2.5-Coder-7B model [10], achieving an execution accuracy of 66.8% on the BIRD development set—surpassing many fine-tuned baselines. In Alpha-SQL, the Text-to-SQL task is formulated as a Monte Carlo Tree Search (MCTS) [1] process, with seven predefined actions. Starting from the root node, each action generates a new node, forming a path to a leaf node that corresponds to one candidate SQL query—completing a single rollout. After 24 rollouts, the final SQL is selected via majority voting based on the consistency of execution results.

Despite its high accuracy, Alpha-SQL suffers from significant inefficiencies—namely high latency and excessive token generation. Through careful implementation and analysis of their codebase, we identify three key factors contributing to this inefficiency:

(1) Leaf nodes sharing the same parent inherit identical reasoning traces (i.e., chains of thought), leading to redundant SQL generation. Even across different parents, partial reasoning paths are frequently reused, reducing diversity and amplifying redundancy.
(2) Node and SQL quality are represented solely by scalar scores. Although these scores are backpropagated along the path after each rollout to guide future node selection, the search process fails to leverage richer historical information—such as which segments of prior reasoning were likely erroneous—to improve search efficiency.
(3) The method requires all 24 rollouts to complete before termination, lacking a flexible early-exit mechanism based on convergence or confidence.

Closely related, MCTS-SQL [45] also adopts MCTS for SQL generation and incorporates a self-evaluation-and-refinement loop to enhance efficiency. However, its design is oversimplified, achieving only 53.0% execution accuracy on the same model (Qwen2.5-Coder-7B), significantly underperforming Alpha-SQL. This gap highlights the need for a more effective and efficient TTS framework.

Building upon these insights, we argue that an ideal, production-ready Text-to-SQL system [4] should satisfy three criteria:

(1) **Zero-shot:** Rely solely on algorithmic enhancements to strengthen the model's intrinsic reasoning capabilities, ensuring generalization across domains and question types.
(2) **Open-source compatible:** Operate on open-source models in an "off-the-shelf" manner, avoiding API calls and mitigating data privacy risks.
(3) **Efficient inference:** Minimize both latency and token consumption while preserving high accuracy.

To this end, we propose **ExpeSQL**, a novel framework that integrates divide-and-conquer, Best-of-N sampling, and self-reflection [29]. As illustrated in Figure 2, the workflow of our method comprises the following stages.

- **Schema Linking:** This step is necessary because the values in a database may be numerous, it is generally impossible to input all of them input the model due to the limitation of the prompt length [32].
- **Divide-and-conquer:** The core idea of divide-and-conquer is decomposing the questions into several simple sub questions and solve them one-by-one [33]. This approach has been proven to be effective in several fileds including Text-to-SQL [23, 39].
- **SQL Selection:** Selecting the SQL query in a pool of candidates.
- **Self-critique:** Evaluating the selected SQL query and decide if another iteration is necessary.
- **Error Diagnosis and Remediation:** When the self-critique step identifies an error, this agent analyzes the reasoning trajectory to pinpoint root causes, generates targeted remediation actions, and stores the diagnostic results in a long-term memory for future reasoning reconstruction.

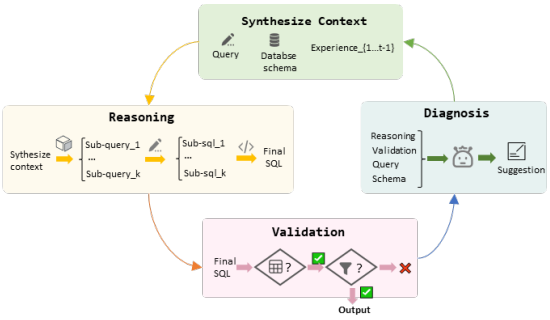

Figure 1: Reasoning loop of ExpeSQL.

ExpeSQL introduces a closed-loop reasoning architecture that combines intra-node diversity preservation, inter-node consensus voting, and self-critique validation with a persistent experience repository. This enables the system to not only generate SQL candidates but also learn from its own reasoning traces, progressively reducing error recurrence over refinement rounds. The key contribution of our approach includes:

(1) We propose ExpeSQL, a novel experience-driven, self-refining framework for Text-to-SQL that integrates decompositional reasoning, execution feedback, and multi-round refinement in a unified pipeline.
(2) We design a self-critique agent and a diagnostic agent that leverage the divide-and-conquer structure to enable fine-grained, modular validation and precise root-cause analysis, significantly enhancing both accuracy and interpretability in Text-to-SQL systems.
(3) We introduce a dual filtering mechanism (intra-node diversity preservation and inter-node consensus voting) and a structured experience repository E that enables the system to learn from past reasoning traces and prevent error recurrence.
(4) Experimental results demonstrate that our method achieves a strong balance between effectiveness and efficiency, delivering high accuracy with controlled computational cost.

With open source model, our approach obtained an execution accuracy of 67.5% on BIRD dev set and achieved a good trade-off between effectiveness and efficiency.

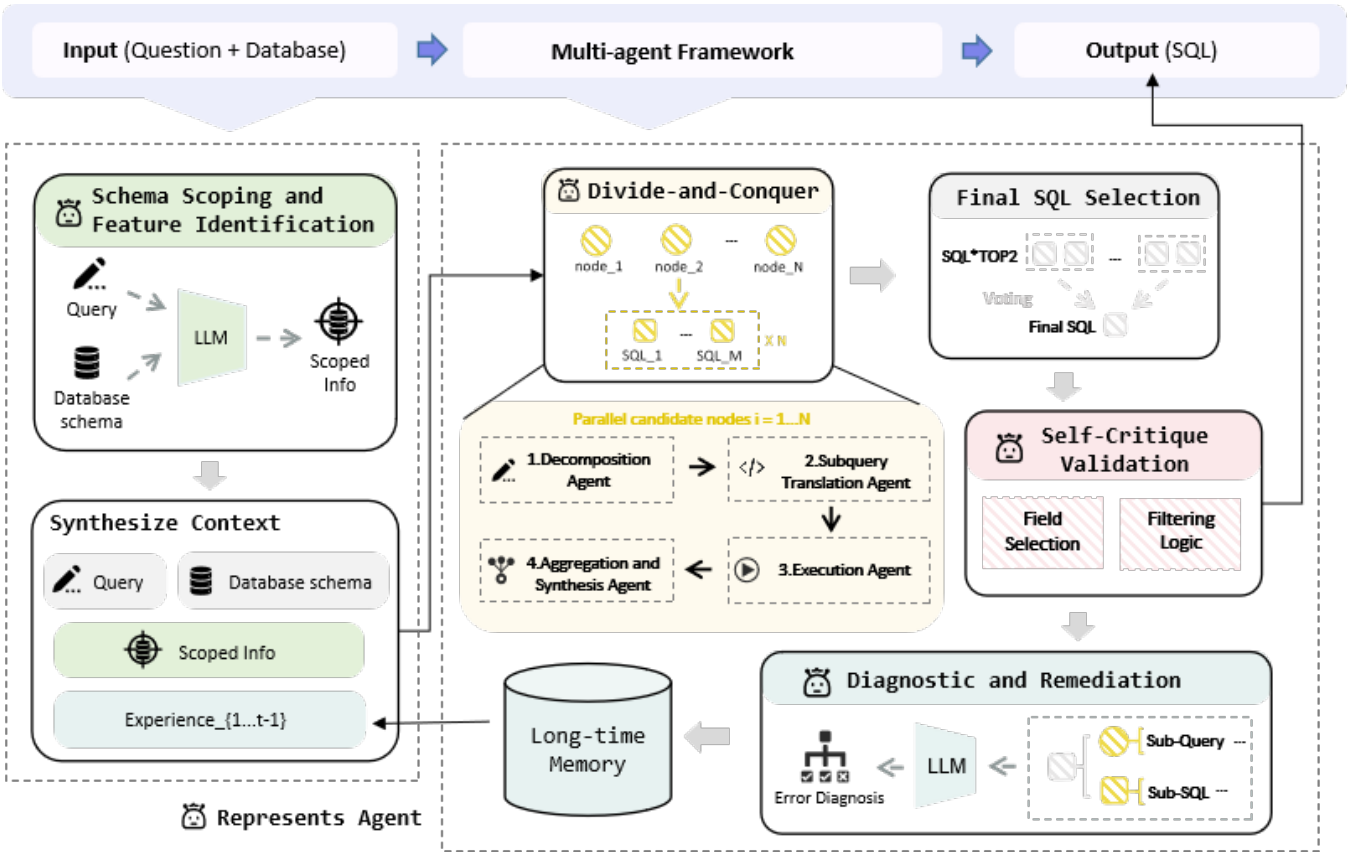

Figure 2: Overview of ExpeSQL: A closed-loop, experience-driven Text-to-SQL system. Each round involves M parallel nodes generating and filtering SQL candidates via decompositional reasoning. Inter-node majority voting yields Σ_t, which is validated and critiqued. If invalid, root causes $\mathcal{E}_t$ and remedies $\mathcal{R}_t$ are extracted and stored in experience repository E which guides future rounds, enabling self-refinement over T iterations.

2 METHOD: EXPESQL

2.1 Overview

We formalize a decompositional reasoning space for Text-to-SQL: given the database schema D and a natural-language query q, the Schema Scoping and Feature Identification Agent first identifies the relevant tables and columns, producing a textual schema rationale p that is incorporated into subsequent reasoning steps.

$$p = \text{Agent}_{\text{scope}}(q, \mathcal{D}) \in \mathcal{P} \tag{1}$$

Next, the Decomposition Agent breaks q into a set of independent sub-questions $S = \{s_i\}$. For each sub-question s_i, the Subquery Translation Agent generates a corresponding sub-SQL query σ_i, which is executed by the Execution Agent to yield a result r_i. These intermediate results are used to construct a final SQL query Σ that integrates all sub-queries, completing a full divide-and-conquer reasoning cycle.

$$\begin{aligned} \Sigma &= \text{Compose}\Big(\big\{\text{Execute}\big(\text{Translate}(s_i)\big)\big\}_{i=1}^{|S|}\Big), \\ S &= \text{Decompose}(q) \end{aligned} \tag{2}$$

Our framework operates sequentially over T refinement rounds, with each round guided by a long-term experience repository E that stores historical reasoning traces and remediation knowledge. In each round t, M independent reasoning nodes execute the complete pipeline in parallel. Crucially, within each node, the final SQL generation process produces K candidate queries to enhance solution space coverage.

Each node then applies **intra-node result-based filtering**: the K candidates are grouped by their execution results, and the two most frequent distinct outcomes are retained. From each of these two result groups, the shortest SQL is selected—yielding exactly two high-quality, diverse candidates per node.

This results in a total of $2M$ candidate SQLs across all nodes. The system then performs **inter-node majority voting** over these $2M$ candidates: the SQL whose execution result appears most frequently is selected as the round-t output Σ_t; in case of a tie, the shortest such SQL is chosen.

$$
\begin{aligned}
&C_t : \text{ all candidate SQLs in round } t,\\
&\Sigma_t \in \arg\max_{\sigma \in C_t} \#\big\{\, \sigma' \in C_t \;:\; \mathrm{Exec}(\sigma') = \mathrm{Exec}(\sigma) \,\big\},\\
&\text{tie-break: pick the shortest such } \sigma.
\end{aligned} \tag{3}
$$

The **Self-Critique Validation Agent** evaluates Σ_t along two axes: (i) *field selection correctness* and (ii) *filtering correctness*. If both pass, Σ_t is emitted as the final output. Otherwise, the **Diagnostic and Remediation Agent** receives the full reasoning chain $\langle q, p, S, \{\sigma_i\}, \{r_i\}, \Sigma_t \rangle$, performs backtracking to localize root causes $\mathcal{E}_t$ (e.g., decomposition bias in s_k or translation/logic error in σ_k), and issues repair suggestions $\mathcal{R}_t$.

$$
\mathrm{Accept}(\Sigma_t) \iff \mathcal{V}_{\text{field}}(\Sigma_t) \;\wedge\; \mathcal{V}_{\text{filter}}(\Sigma_t) \tag{4}
$$

$$
\big(\mathcal{E}_t,\ \mathcal{R}_t\big) \;=\; \mathrm{Diagnose}(\langle q,\ p,\ S,\ \{\sigma_i\},\ \{r_i\},\ \Sigma_t \rangle) \tag{5}
$$

These structured artifacts—including p, s_i, σ_i, r_i, Σ_t, self-critique judgments, error taxonomy, and remediation actions—are persisted into E. In subsequent rounds, E is proactively retrieved to guide decomposition, prioritize robust candidates, and prevent recurrence of prior errors.

$$
\begin{aligned}
&\text{Let } \Phi_t = \langle q, \mathcal{D}, p, S, \{\sigma_i\}, \{r_i\}, \Sigma_t \rangle,\\
&\text{Update experience repository:}\\
&E \leftarrow E \cup \{\Phi_t, \mathcal{E}_t, \mathcal{R}_t\}\,.
\end{aligned} \tag{6}
$$

The process continues sequentially until a valid SQL is produced or the maximum round T is reached. The final output Σ^* is the last valid SQL generated, ensuring progressive refinement grounded in historical experience.

2.2 Agents

ExpeSQL employs an agentic Text-to-SQL pipeline [17], where specialized LLM agents handle schema scoping, decomposition, translation, execution, and validation. This design ensures modularity, execution-level verifiability, and iterative refinement through a persistent experience repository E that captures errors, diagnostics, and fixes for reuse in subsequent rounds.

Schema Scoping and Feature Identification Agent. This agent initiates the reasoning pipeline by performing a coarse-grained relevance analysis over the database schema. Given a natural language question, it prompts the large language model to identify and output the subset of tables and their relevant columns that are potentially involved in answering the query. By focusing only on semantically related schema elements, this step effectively narrows the search space for subsequent agents, reducing noise from irrelevant fields and improving both efficiency and accuracy downstream [3].

Agents of Divide-and-Conquer. The **Decomposition Agent**, **Subquery Translation Agent**, and **Execution Agent** collectively realize the core reasoning loop of ExpeSQL. Given the scoped schema, the Decomposition Agent breaks the natural language query into a set of independent, semantically self-contained subquestions $S = \{s_i\}$. For each s_i, the Subquery Translation Agent generates a corresponding sub-SQL σ_i using schema-aware prompting and domain-specific templates, ensuring syntactic correctness and semantic alignment. The Execution Agent then executes each σ_i against the database to obtain a concrete result r_i, forming a verifiable triplet $\langle s_i, \sigma_i, r_i \rangle$. This tight loop—decompose, translate, execute—not only enables early validation of partial logic but also generates actionable feedback (e.g., empty results, schema mismatches) that informs downstream aggregation and self-critique. By modularizing these steps, the pipeline ensures traceability, reduces error propagation, and supports parallel exploration across multiple reasoning paths [42].

Aggregation and Synthesis Agent. This agent is responsible for constructing the final SQL query by integrating the intermediate sub-SQLs $\{\sigma_i\}$ and their executed results $\{r_i\}$ into a coherent, executable statement. Taking as input the set of verified sub-queries and their observed outcomes, it performs semantic synthesis—aligning projections, aggregations, joins, and filters—to assemble a globally consistent SQL query Σ that reflects the original intent [4].

Crucially, within each reasoning node, the agent generates K candidate queries through controlled perturbation of key components (e.g., join sequences, filtering conditions, aggregation scopes, and projection lists). This Best-of-K generation enables exploration of alternative query structures while grounding synthesis in execution-verified sub-results. The resulting candidates are executed, and the two with the most frequent distinct outcomes are retained, selecting the shortest SQL per outcome for downstream voting.

By centralizing the synthesis process, the agent ensures syntactic completeness and semantic coherence, while enabling explicit feedback between local sub-question resolutions and global query construction—effectively bridging decomposed reasoning to final answer generation [28].

Self-Critique Validation Agent. This agent performs structured, execution-aware validation of candidate SQL queries before finalization [41]. Given a synthesized query Σ, it evaluates correctness along two orthogonal axes: (i) *field selection*, verifying the alignment of projected measures and dimensions with the natural language intent, including correct use of aggregation functions (e.g., `COUNT`, `SUM`), proper aliasing, `DISTINCT` semantics, and validity of selected columns with respect to the database schema; and (ii) *filtering logic*, assessing the validity of `WHERE` and `HAVING` predicates, proper handling of `NULL` values, and avoidance of erroneous join conditions misused as filters. The validation is grounded in both syntactic structure and semantic execution: the agent checks whether the query executes without error and whether intermediate or final results exhibit expected patterns. If Σ passes both criteria, it is emitted as the final answer. Otherwise, the query and its full reasoning chain are forwarded to the Diagnostic and Remediation Agent for deep analysis, enabling failure-aware refinement rather than blind rejection [46].

Diagnostic and Remediation Agent. While some iterative refinement methods attempt to correct logical errors [27, 31, 45], they typically rely on in-place updates guided by coarse execution feedback, leading to local adjustments without systematic diagnosis of root causes. This approach risks incomplete or unstable fixes—such as over-correction (e.g., altering correct conditions) or cascading errors—due to the lack of a reconstructed reasoning trajectory. For example, in work [2], which leverages powerful models such as GPT-4o and DeepSeek-R1, iterative refinement yields less than a 0.8% improvement in overall accuracy.

By integrating with the divide-and-conquer framework, the Diagnostic and Remediation Agent adopts a self-reflective mechanism that generates fine-grained, sub-problem-level feedback—pinpointing errors in specific sub-queries or decomposition steps—and stores these insights in a long-term memory module to guide a complete reconstruction of the reasoning trajectory in subsequent iterations, rather than performing in-place SQL edits. Upon receiving a failed reasoning trace, this agent performs causal backtracking to localize and characterize errors within the pipeline [34]. It takes as input the complete structured context $\langle q, S, \{\sigma_i\}, \{r_i\}, \Sigma\rangle$, reconstructs the execution flow, and identifies root causes such as decomposition bias (e.g., missing or over-partitioned sub-questions s_k), translation inaccuracies (e.g., incorrect joins or filters in σ_k), or logical inconsistencies (e.g., type mismatches, semantic drift). Based on this diagnosis, it generates targeted repair suggestions—such as reformulating a sub-question, adjusting join logic, or revising aggregation scope—and persists the entire artifact tuple (s_i, σ_i, r_i, Σ, $\mathcal{E}_t$, $\mathcal{R}_t$) into the long-term experience repository E. This structured memory enables continual learning [38]: in subsequent rounds, the system proactively retrieves relevant past failures and fixes to guide decomposition, prioritize robust paths, and avoid recurrent errors, thereby closing the loop between execution, critique, and improvement.

2.3 Context Engineering And Experience Learning

To facilitate context management [22], we adopt a short- and long-term memory mechanism [47]. At the beginning of each iteration, an **action node** is created to encapsulate the full divide-and-conquer reasoning trace. This node includes the decomposed sub-questions $S = \{s_i\}$, their corresponding sub-SQLs $\{\sigma_i\}$, and the execution results $\{r_i\}$—organized as a sequence of verifiable reasoning steps, forming the short-term memory of the current round. Additionally, the action node stores metadata from the current iteration, including the final SQL query, its execution outcome, the self-critique result, and, if validation fails, the diagnostic output from causal backtracking.

When self-critique fails and a new iteration is initiated, the current action node becomes a child of the previous action node, forming a temporally ordered chain. This hierarchical linking across iterations constitutes the system's long-term memory, preserving historical reasoning traces and error recovery contexts. At the start of each new cycle, this long-term memory is traversed, and relevant segments are selectively incorporated into the context to guide subsequent reasoning. The memory is reconstructed in a sequential format:

$$\texttt{round}_1 : (s_i,\ \sigma_i,\ r_i,\ \Sigma,\ \mathcal{E}_t,\ \mathcal{R}_t) \xrightarrow{\text{failure}} \texttt{round}_2 : (\ldots) \xrightarrow{\text{failure}} \cdots \xrightarrow{\text{success}} \texttt{round}_k$$

Agents that leverage long-term memory—including the **Schema Scoping and Feature Identification Agent**, the **Decomposition Agent**, and the **Aggregation and Synthesis Agent**—use this accumulated experience to inform schema relevance analysis, subproblem decomposition, and query synthesis, respectively.

2.4 Preprocessing

We adopt the schema-linking methodology of [14, 35]. For each database, we first preprocess textual column values using MinHash [6] to produce compact signatures, which we store locally. Given a user query, we extract salient keywords with a LLM and apply locality-sensitive hashing to efficiently retrieve candidate values from the precomputed signatures. We then filter candidates using edit-distance and semantic-similarity thresholds, where semantic similarity is computed with OpenAI's text-embedding-3-large model. The resulting matches are incorporated into the database-schema prompt, thereby providing contextual grounding for the LLM during SQL generation.

3 EXPERIMENTS

Datasets and Metrics. We evaluate our model on two widely used benchmarks: BIRD [16] and Spider [44] development sets, containing 1,534 and 1,034 question-SQL pairs, respectively. BIRD is more challenging than Spider due to its diverse domains and complex queries, providing a stronger evaluation of Text-to-SQL methods. We use Execution Accuracy (EX) as the primary metric, which measures whether generated SQL queries produce the same results as the gold queries. We also compare token usage with Alpha-SQL to evaluate computational efficiency.

Implementation details. We use instruction-tuned Qwen-Coder models, including Qwen2.5-Coder-14B-Instruct, Qwen2.5-Coder-32B-Instruct, and Qwen3-Coder-30B-Instruct [10, 37]. In the Best-of-N framework, we generate three reasoning nodes per iteration, with each node producing eight candidate SQL queries for voting. The maximum iteration number is set to three, allowing up to two refinement steps after error detection.

4 RESULTS

BIRD dataset. As shown in Table 1, fine-tuning significantly improves model performance on the BIRD dataset. Nevertheless, ExpeSQL outperforms several existing methods—even those based on powerful closed-source models. For instance, using the Qwen3-Coder-30B model, our approach achieves higher accuracy than CHESS-SQL (which uses GPT-4-Turbo) and Distillery (which uses GPT-4o). When training is applied to open-source models, only XiYan-SQL with M-Schema representation and Reasoning-SQL—trained via reinforcement learning—surpass ExpeSQL in performance. Notably, when XiYan-SQL adopts DDL-based schema encoding (the same representation used in our method), its performance drops below that of ExpeSQL when both are instantiated with the Qwen2.5-Coder-32B model.

In comparison with other methods in the same category—i.e., zero-shot approaches using open-source models—our method demonstrates clear advantages, substantially outperforming MCTS-SQL, ROUTE, and Distillery. The relatively small number of methods in this category reflects, to some extent, the inherent difficulty of achieving strong performance without any training.

When compared to Alpha-SQL, our method lags behind across all three model variants, though the performance gap narrows as the base model improves. While there remains a gap in execution accuracy, TTS methods are typically evaluated along two dimensions: computational cost and model performance. Experimental

Table 1: Comparison of state-of-the-art Text-to-SQL methods on the BIRD dev set.

Method	Inference Model	Selection Model	Zero-shot	Open-Source	EX (%)
CHESS-SQL [35]	Deepseek-Coder-33B	GPT-4-Turbo	No	No	65.0
Distillery [21]	GPT-4o	—	No	No	67.2
CHASE-SQL [23]	Gemini-1.5-Pro	Gemini-1.5-Flash	No	No	73.0
XiYan-SQL [19]	Unreported	Unreported	No	No	73.3
DAIL-SQL [8]	GPT-4	Majority Voting	Yes	No	55.9
SuperSQL [13]	GPT-4	Majority Voting	Yes	No	58.5
MAC-SQL [39]	GPT-4	Iterative Refinement	Yes	No	59.4
Gen-SQL [31]	GPT-4	Iterative Refinement	Yes	No	59.8
RSL-SQL [2]	GPT-4o	Iterative Refinement	Yes	No	67.2
DTS-SQL [25]	DeepSeek-7B	—	No	Yes	55.8
SFT CodeS [15]	CodeS-15B	—	No	Yes	58.5
ROUTE (Multi-task + FT) [27]	Qwen2.5-Coder-14B	Iterative Refinement	No	Yes	60.9
CHESS-SQL [35]	Deepseek-Coder-33B	LLaMA3-70B	No	Yes	61.5
XiYan-SQL@DDL [19]	Qwen2.5-Coder-32B	Qwen2.5-Coder-32B	No	Yes	62.3
XiYan-SQL@M-Schema [19]	Qwen2.5-Coder-32B	Qwen2.5-Coder-32B	No	Yes	67.0
Reasoning-SQL [26]	Qwen2.5-Coder-14B	—	No	Yes	72.3
MCTS-SQL [45]	Qwen2.5-Coder-7B	Iterative Refinement	Yes	Yes	53.6
ROUTE (Multi-task only) [27]	Qwen2.5-Coder-14B	Iterative Refinement	Yes	Yes	56.3
Distillery [21]	Llama-3.1-405B	—	Yes	Yes	59.2
Alpha-SQL* [14]	Qwen2.5-Coder-14B	Majority Voting	Yes	Yes	64.1
Alpha-SQL* [14]	Qwen2.5-Coder-32B	Majority Voting	Yes	Yes	64.4
Alpha-SQL* [14]	Qwen3-Coder-30B-A3B-Instruct	Majority Voting	Yes	Yes	68.2
ExpeSQL (Ours)	Qwen2.5-Coder-14B	Iterative Refinement	**Yes**	**Yes**	**61.4**
ExpeSQL (Ours)	Qwen2.5-Coder-32B	Iterative Refinement	**Yes**	**Yes**	**63.5**
ExpeSQL (Ours)	Qwen3-Coder-30B-A3B-Instruct	Iterative Refinement	**Yes**	**Yes**	**67.5**

***EX**: Execution accuracy on BIRD dev set. * Here we report results obtained using our own evaluation script, which we believe provides a more reasonable assessment; see Appendix A.1 for detailed justification.*

results show that, compared to Alpha-SQL, our Qwen2.5-Coder-32B model incurs only a 0.9% lower accuracy, while significantly reducing token generation by 87% and inference latency by 96%. This substantial reduction translates into markedly lower inference latency and decreased API costs in real-world deployment, demonstrating a favorable trade-off between efficiency and performance. Detailed experimental settings and further analysis of the results are provided in Section A.2.

Another observation worth noting is that all methods employing *Iterative Refinement*, except for RSL-SQL based on GPT-4o, achieve relatively limited performance. For instance, ROUTE—even with fine-tuning on the Qwen2.5-Coder-14B model—still underperforms compared to ExpeSQL. This suggests that simple error detection followed by direct SQL refinement is an overly simplistic strategy with significant limitations.

In contrast, our method does not directly modify the generated SQL. Instead, it identifies the root cause of the error and, in the subsequent iteration, leverages prior experience to reformulate the query understanding and generation process. This more principled approach to iterative reasoning leads to substantially improved performance.

Spider dataset. ExpeSQL with the Qwen2.5-Coder-14B model outperforms DAIL-SQL and ZeroNL2SQL, both of which are based on GPT-4. Notably, despite sharing similar components—such as multi-task learning and iterative refinement—our method achieves significantly better performance on Qwen2.5-Coder-14B than the non-fine-tuned ROUTE, demonstrating the effectiveness of our design choices.

Table 2: Comparison of state-of-the-art Text-to-SQL methods on the Spider dev set.

Method	Inference Model	Selection Model	Zero-shot	Open-Source	EX (%)
DAIL-SQL [8]	GPT-4	Majority Voting	Yes	No	83.6
ZeroNL2SQL [7]	GPT-4	-	Yes	No	84.0
MAC-SQL [39]	GPT-4	Majority Voting	Yes	No	86.8
SuperSQL [13]	GPT-4	Majority Voting	Yes	No	87.0
SFT CodeS [15]	CodeS-15B	-	No	Yes	84.9
ROUTE (Multi-task + FT) [27]	Qwen2.5-Coder-14B	Iterative Refinement	No	Yes	87.3
ROUTE (Multi-task only) [27]	Qwen2.5-Coder-14B	Iterative Refinement	Yes	Yes	80.0
Alpha-SQL* [14]	Qwen2.5-Coder-7B	Majority Voting	Yes	Yes	84.0
Alpha-SQL* [14]	Qwen2.5-Coder-14B	Majority Voting	Yes	Yes	87.0
ExpeSQL (Ours)	Qwen2.5-Coder-14B	Iterative Refinement	**Yes**	**Yes**	**84.3**
ExpeSQL (Ours)	Qwen3-Coder-30B-A3B-Instruct	Iterative Refinement	**Yes**	**Yes**	**86.2**

***EX**: Execution accuracy on Spider dev set. * Results for Alpha-SQL are directly taken from the original paper without re-evaluation. Due to differences in evaluation protocols, the actual accuracy under our setup may be lower than reported.*

4.1 Ablation Studies

We conduct an ablation study to evaluate the contribution of each component in our framework. Table 3 summarizes the results on the BIRD-dev dataset.

The baseline model, Qwen32B-Coder-32B with preprocessed schema information, achieves 53.3% accuracy. Adding schema linking improves performance to 55.2% (+1.9%), demonstrating the importance of identifying relevant database elements.

The Divide-and-Conquer strategy further increases accuracy to 59.1% (+3.9%), showing that query decomposition facilitates more effective reasoning. Adding self-reflection improves accuracy to 62.2% (+3.1%), indicating the benefit of iterative refinement.

Finally, Best-of-N achieves 63.5% accuracy (+1.3%), confirming its effectiveness in selecting better outputs from multiple candidates.

Overall, each component provides consistent improvements, demonstrating the effectiveness of our framework on the challenging BIRD-dev dataset.

Table 3: Progressive performance on BIRD-dev

Method	Accuracy (%)
Qwen32B-Coder-32B + Preprocessed Schema	53.3
+Schema Linking	55.2 (+1.9)
+Divide-and-Conquer	59.1 (+3.9)
+Self-Reflection	62.2 (+3.1)
+Best-of-N	63.5 (+1.3)

5 LIMITATIONS AND FUTURE WORK

Although our model achieves strong performance on both datasets, several limitations remain:

(1) **Diversity of SQL Generation**: Increasing reasoning nodes beyond three does not improve performance. While divide-and-conquer enables error localization, a single decomposition strategy may limit SQL diversity. Future work could explore more diverse generation methods.

(2) **Utilization of Experience**: Intermediate SQL candidates discarded during voting may still contain valuable information. Better exploitation of these candidates could improve future SQL generation.

(3) **Alignment between SQL and Natural Language**: Our method depends on accurate alignment between questions and SQL queries. Improving error verification beyond column and condition checks may further enhance performance.

These limitations provide opportunities for future research toward more diverse and experience-driven Text-to-SQL systems.

REFERENCES

[1] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[2] Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073* (2024).

[3] Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2025. Reliable Text-to-SQL with Adaptive Abstention. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–30.

[4] Song Cheng, Qiannan Cheng, Linbo Jin, Lei Yi, and Guannan Zhang. 2025. SQLord: A Robust Enterprise Text-to-SQL Solution via Reverse Data Generation and Workflow Decomposition. In *Companion Proceedings of the ACM on Web Conference 2025*. 919–923.

[5] Team Cohere, Arash Ahmadian, Marwan Ahmed, Jay Alammar, Milad Alizadeh, Yazeed Alnumay, Sophia Althammer, Arkady Arkhangorodsky, Viraat Aryabumi, Dennis Aumiller, et al. 2025. Command a: An enterprise-ready large language model. *arXiv preprint arXiv:2504.00698* (2025).

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (Brooklyn, New

York, USA) *(SCG '04)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[7] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot nl2sql. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2750–2763.

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

[9] Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking. *arXiv preprint arXiv:2501.04519* (2025).

[10] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[12] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467* (2024).

[13] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The dawn of natural language to sql: Are we fully ready? *arXiv preprint arXiv:2406.01265* (2024).

[14] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *arXiv preprint arXiv:2502.17248* (2025).

[15] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[16] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2023), 42330–42357.

[17] Jiahui Li, Tongwang Wu, Yuren Mao, Yunjun Gao, Yajie Feng, and Huaizhong Liu. 2025. SQL-Factory: A Multi-Agent Framework for High-Quality and Large-Scale SQL Generation. *arXiv preprint arXiv:2504.14837* (2025).

[18] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025. Nl2sql-bugs: A benchmark for detecting semantic errors in nl2sql translation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5662–5673.

[19] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, et al. 2025. XiYan-SQL: A Novel Multi-Generator Framework For Text-to-SQL. *arXiv preprint arXiv:2507.04701* (2025).

[20] Shuai Lyu, Haoran Luo, Ripeng Li, Zhonghong Ou, Jiangfeng Sun, Yang Qin, Xiaoran Shang, Meina Song, and Yifan Zhu. 2025. SQL-o1: A Self-Reward Heuristic Dynamic Search Method for Text-to-SQL. *arXiv preprint arXiv:2502.11741* (2025).

[21] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702* (2024).

[22] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. 2025. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334* (2025).

[23] 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. *arXiv preprint arXiv:2410.01943* (2024).

[24] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 36339–36348. https://proceedings.neurips.cc/paper_files/paper/2023/file/72223cc66f63ca1aa59edaec1b3670e6-Paper-Conference.pdf

[25] Mohammadreza Pourreza and Davood Rafiei. 2024. Dts-sql: Decomposed text-to-sql with small large language models. *arXiv preprint arXiv:2402.01117* (2024).

[26] Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, Sercan Arik, et al. 2025. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql. *arXiv preprint arXiv:2503.23157* (2025).

[27] Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2024. ROUTE: Robust multitask tuning and collaboration for Text-to-SQL. *arXiv preprint arXiv:2412.10138* (2024).

[28] Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. 2025. SHARE: An SLM-based Hierarchical Action CorREction Assistant for Text-to-SQL. *arXiv preprint arXiv:2506.00391* (2025).

[29] Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682* (2024).

[30] Lei Sheng and Shuai-Shuai Xu. 2025. CSC-SQL: Corrective Self-Consistency in Text-to-SQL via Reinforcement Learning. *arXiv preprint arXiv:2505.13271* (2025).

[31] Jie Shi, Bo Xu, Jiaqing Liang, Yanghua Xiao, Jia Chen, Chenhao Xie, Peng Wang, and Wei Wang. 2025. Gen-SQL: Efficient Text-to-SQL by bridging natural language question and database schema with pseudo-schema. In *Proceedings of the 31st International Conference on Computational Linguistics*. 3794–3807.

[32] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. 2025. Automatic Metadata Extraction for Text-to-SQL. *arXiv preprint arXiv:2505.19988* (2025).

[33] Douglas R Smith. 1985. The design of divide and conquer algorithms. *Science of Computer Programming* 5 (1985), 37–58.

[34] Oleg Somov and Elena Tutubalina. 2025. Confidence estimation for error detection in text-to-sql systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 25137–25145.

[35] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755* (2024).

[36] Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[37] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[38] Gido M Van de Ven and Andreas S Tolias. 2019. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734* (2019).

[39] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2023. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242* (2023).

[40] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–24.

[41] Bo Xu, Shufei Li, Hongyu Jing, Ming Du, Hui Song, Hongya Wang, and Yanghua Xiao. 2025. Boosting Text-to-SQL through Multi-grained Error Identification. In *Proceedings of the 31st International Conference on Computational Linguistics*. 4282–4292.

[42] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[43] Zerui Yang, Yuwei Wan, Siyu Yan, Yudai Matsuda, Tong Xie, Bram Hoex, and Linqi Song. 2025. DrugMCTS: a drug repurposing framework combining multi-agent, RAG and Monte Carlo Tree Search. *arXiv preprint arXiv:2507.07426* (2025).

[44] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

[45] Shuozhi Yuan, Liming Chen, Miaomiao Yuan, Jin Zhao, Haoran Peng, and Wenming Guo. 2025. Mcts-sql: An effective framework for text-to-sql with monte carlo tree search. *arXiv preprint arXiv:2501.16607* (2025).

[46] Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. Optimizing Reasoning for Text-to-SQL with Execution Feedback. In *Findings of the Association for Computational Linguistics: ACL 2025*. 19206–19218.

[47] Kai Zhang, Yangyang Kang, Fubang Zhao, and Xiaozhong Liu. 2023. Llm-based medical assistant personalization with short-and long-term memory coordination. *arXiv preprint arXiv:2309.11696* (2023).

[48] Qinggang Zhang, Hao Chen, Junnan Dong, Wentao Li, Feiran Huang, and Xiao Huang. 2025. Structure-Guided Large Language Models for Text-to-SQL Generation. (2025).

[49] Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. Elliesql: Cost-efficient text-to-sql with complexity-aware routing. *arXiv preprint arXiv:2503.22402* (2025).

A APPENDIX

A.1 On the Treatment of Empty Query Results in Evaluation

In our attempt to reproduce the results of AlphaSQL, we observed a discrepancy between our initial evaluation and those reported by the authors. Upon obtaining their evaluation script, we successfully reproduced their results and found that their protocol automatically discards SQL predictions yielding empty results during majority voting, falling back to the next most frequent non-empty result. While this adjustment enables faithful reproduction of their published numbers, we argue that such a design implicitly assumes—often incorrectly—that all valid SQL queries must return non-empty results. In real-world applications, however, it is entirely plausible for a semantically correct query to return an empty result (e.g., "find customers with 10+ orders last month" in a newly launched business). We also examined several representative Text-to-SQL methods, including those based on iterative refinement or majority voting [40] [24] [39] [13], and found no evidence of such empty-result filtering in their public implementations or evaluation protocols. We therefore adopt a more principled evaluation protocol: we only filter out SQL queries that fail to execute or raise parsing/execution errors, but preserve those that return empty results upon successful execution. To ensure a fair and realistic comparison, we report the performance of AlphaSQL on the BIRD development set using our evaluation protocol.

A.2 Efficiency and Performance Analysis

Experiment Settings. We measure the token generation cost and inference latency of both models on the CHESS-SDS [35] dataset. For the experiments involving Alpha-SQL and ExpeSQL, we deploy Qwen32B-Coder-32B locally using vLLM [11] on four Nvidia GeForce RTX 4090 GPUs, each with 24GB of memory. During evaluation, we process one question at a time and record the total execution time and the total number of generated tokens per question. The final results are reported as averages over all questions.

It is important to note that for Alpha-SQL, the reported latency does not include the time required for majority voting. Specifically, timing stops immediately after all candidate SQL queries are generated. Therefore, the actual end-to-end inference time would be longer than reported. In contrast, for ExpeSQL, majority voting is an integral part of the inference pipeline, and its computational cost is included in the total latency.

Result. As shown in Table 4, compared to Alpha-SQL, our method not only improves accuracy but also significantly reduces computational cost and inference latency. Specifically, our approach reduces token generation by 87% and cuts inference latency by 96%. This dramatic improvement stems from the high degree of parallelism in our algorithm—except for the self-critique and error diagnosis stages, all other steps are executed in parallel. In contrast, Alpha-SQL only parallelizes the expansion phase. As a result, our method achieves highly efficient inference while maintaining strong performance, outperforming all models in the comparison except for Gemini-2.0-Flash-Thinking-Exp, where it is slightly behind.

We further evaluated our method on Llama-3.1-8B [36], achieving a substantial improvement of approximately 24% over the baseline. This demonstrates the strong generalizability and plug-and-play capability of our approach.

A.3 Related Work

A.3.1 Typical Paradigms in Text-to-SQL. Modern Text-to-SQL systems follow a three-stage pipeline: schema linking [32], candidate generation [49], and candidate selection [30]. Schema linking aligns natural language with database elements to reduce ambiguity, evolving from rule-based matching [20] to fine-tuned retrievers for complex schemas [27]. In the second stage, SQL candidates are generated via single-path decoding [39] or multi-path reasoning [42], often using modular decomposition [27] or diverse prompting [12]. The final stage selects the best candidate using execution feedback [14] or learned rankers [23].

State-of-the-art systems like XiYan-SQL [19] combine multi-path generation with fine-tuned components for high accuracy, but rely heavily on labeled data and are brittle under schema changes—limiting adaptability in dynamic environments. Moreover, many depend on closed-source LLMs [40], introducing latency, cost, and privacy concerns in production.

A.3.2 TTS for Text-to-SQL. To overcome the limitations of fine-tuning, recent work has explored TTS—a paradigm that enhances reasoning performance at inference time without parameter updates. TTS methods typically generate diverse SQL candidates through strategic search [20] (e.g., self-consistency, chain-of-thought variation, or tree search) and select the final output based on execution feedback or majority voting.

MAC-SQL [39] adopts a divide-and-conquer framework to decompose complex questions into sub-queries, enabling structured reasoning over large databases. However, it relies on closed-source models such as GPT-4 and achieves only 59.4% execution accuracy on the BIRD dev set. In contrast, Alpha-SQL [14] operates entirely with open-source LLMs under the test-time scaling paradigm and requires no fine-tuning, achieving state-of-the-art performance among such methods.

Nonetheless, both approaches illustrate a persistent challenge in Text-to-SQL: the difficulty of balancing *accuracy* and *efficiency*. MAC-SQL [39], despite using powerful proprietary models, fails to deliver high accuracy, suggesting that decomposition alone is insufficient for complex reasoning. Alpha-SQL [14] achieves strong accuracy but at the cost of high computational overhead due to inefficient search strategies. This trade-off reveals a critical gap—existing methods either sacrifice performance for practicality or efficiency for accuracy—leaving a need for systems that achieve both high correctness and deployable efficiency.

To bridge this gap, we propose **ExpeSQL**, a highly efficient and accurate Text-to-SQL framework that achieves strong performance without sacrificing inference speed or token efficiency. By integrating decomposition-based reasoning, experience-guided search, and parallel Best-of-N sampling, our method navigates the accuracy-efficiency trade-off more effectively than prior work—enabling robust, deployable Text-to-SQL in real-world settings.

Table 4: Comparison of Baseline LLMs on the CHESS-SDS Set

Model	EX (%)	Tokens/Query	Time/Query (s)
Deepseek-R1	50.3	–	–
GPT-4o	53.7	–	–
Gemini-2.0-Flash-Thinking-Exp	60.8	–	–
Qwen32B-Coder-32B	49.0	282	8
+ Alpha-SQL	58.5	111778	1628
+ **ExpeSQL**	59.9	14510	66
Llama-3.1-8B	21.1	–	–
+ **ExpeSQL**	45.0	–	–

Algorithm 1 Experience-Guided Divide-and-Conquer Text-to-SQL with Iterative Refinement

Require: Database schema $\mathcal{D}$, Natural language query q, Max refinement rounds T, Number of parallel nodes M, Candidates per node K
Ensure: Final SQL query Σ^*

Initialize experience repository $E \leftarrow \emptyset$ ▷ Stores past traces & remedies
for $t = 1$ to T **do**
  $\textsc{SchemaScoping}(q, \mathcal{D}) \rightarrow p$ ▷ Relevant tables/columns
  $\textsc{Decomposition}(q, p, E) \rightarrow S = \{s_i\}$ ▷ Sub-questions
  for each node $m = 1$ to M **in parallel do**
    for each sub-question $s_i \in S$ **do**
      $\textsc{SubqueryTranslation}(s_i, p, \mathcal{D}) \rightarrow \sigma_i$
      $\textsc{Execution}(\sigma_i, \mathcal{D}) \rightarrow r_i$ ▷ Intermediate result
    end for
    $\textsc{AggregationAndSynthesis}(S, \{\sigma_i\}, \{r_i\}, p, E) \rightarrow \{\Sigma_t^{(k)}\}_{k=1}^{K}$ ▷ Generate K SQL queries
    Intra-node filtering:
    Group $\{\Sigma_t^{(k)}\}$ by execution result $r^{(k)}$
    Retain top-2 most frequent **distinct** results
    for each of the two result groups **do**
      Select shortest SQL $\Sigma_{t,m}^{(1)}, \Sigma_{t,m}^{(2)}$
    end for
  end for
  Collect all node outputs: $C_t = \{\Sigma_{t,m}^{(1)}, \Sigma_{t,m}^{(2)}\}_{m=1}^{M}$
  Inter-node majority voting:
  Group C_t by execution result
  Select most frequent result; break ties by shortest SQL
  $\Sigma_t \leftarrow$ corresponding SQL query
  Self-Critique Validation$(\Sigma_t, q, \mathcal{D})$ ▷ Uses $\mathcal{D}$ and q
  if $\textsc{ValidateFieldSelection}(\Sigma_t, q, \mathcal{D}) \wedge \textsc{ValidateFilteringLogic}(\Sigma_t, q, \mathcal{D})$ **then**
    return $\Sigma^* \leftarrow \Sigma_t$ ▷ Final output
  else
    $\Phi_t \leftarrow \langle q, \mathcal{D}, p, S, \{\sigma_i\}, \{r_i\}, \Sigma_t \rangle$
    $\textsc{DiagnosticAndRemediation}(\Phi_t) \rightarrow$ root causes $\mathcal{E}_t$ and remedies $\mathcal{R}_t$
    Update $E \leftarrow E \cup \{\Phi_t, \mathcal{E}_t, \mathcal{R}_t\}$
  end if
end for
return $\Sigma^* \leftarrow$ last valid Σ_t (or $\emptyset$ if none)