DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework

BOYAN LI, The Hong Kong University of Science and Technology (Guangzhou), China
CHONG CHEN, Huawei Technologies Co., Ltd., China
ZHUJUN XUE, Huawei Technologies Co., Ltd., China
YINAN MEI, Huawei Technologies Co., Ltd., China
YUYU LUO*, The Hong Kong University of Science and Technology (Guangzhou), China

Large language models (LLMs) have advanced Text-to-SQL, yet existing solutions still fall short of system-level reliability. The limitation is not merely in individual modules – *e.g.,* schema linking, reasoning, and verification – but more critically in the lack of structured orchestration that enforces correctness across the entire workflow. This gap motivates a paradigm shift: treating Text-to-SQL not as free-form language generation but as a software-engineering problem that demands structured, verifiable orchestration. We present DeepEye-SQL, a software-engineering-inspired framework that reframes Text-to-SQL as the development of a small software program, executed through a verifiable process guided by the Software Development Life Cycle (SDLC). DeepEye-SQL integrates four synergistic stages: it grounds user intent through robust schema linking, enforcing relational closure; enhances fault tolerance with N-version SQL generation; ensures deterministic verification via a "Syntax-Logic-Quality" tool-chain that intercepts errors pre-execution; and introduces confidence-aware selection that leverages execution-guided adjudication to resolve ambiguity beyond simple majority voting. Leveraging open-source MoE LLMs (~30B total, ~3B activated parameters) without any fine-tuning, DeepEye-SQL achieves 73.5% execution accuracy on BIRD-Dev, 75.07% on the official BIRD-Test leaderboard, and 89.8% on Spider-Test, outperforming state-of-the-art solutions that rely on larger models or extensive training. This highlights that principled orchestration, rather than LLM scaling alone, is key to achieving system-level reliability in Text-to-SQL.

CCS Concepts: • **Information systems → Structured Query Language**.

Additional Key Words and Phrases: Text-to-SQL, Databases, Large language models

ACM Reference Format:
Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2026. DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 158 (June 2026), 28 pages. https://doi.org/10.1145/3802035

1 Introduction

Text-to-SQL is a task that converts natural-language questions into SQL queries over a database [2, 22, 25–29, 64, 65]. Large language models (LLMs) [45, 51, 53, 54, 59, 63] have substantially advanced Text-to-SQL, achieving strong results on benchmarks such as Spider [57] and BIRD [21]. For

*Yuyu Luo is the corresponding author.

Authors' Contact Information: Boyan Li, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, bli303@connect.hkust-gz.edu.cn; Chong Chen, Huawei Technologies Co., Ltd., Shenzhen, China, chenchong55@huawei.com; Zhujun Xue, Huawei Technologies Co., Ltd., Shenzhen, China, xuezhujun@huawei.com; Yinan Mei, Huawei Technologies Co., Ltd., Shenzhen, China, yinan.mei@huawei.com; Yuyu Luo, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, yuyuluo@hkust-gz.edu.cn.

ACM 2836-6573/2026/6-ART158
https://doi.org/10.1145/3802035

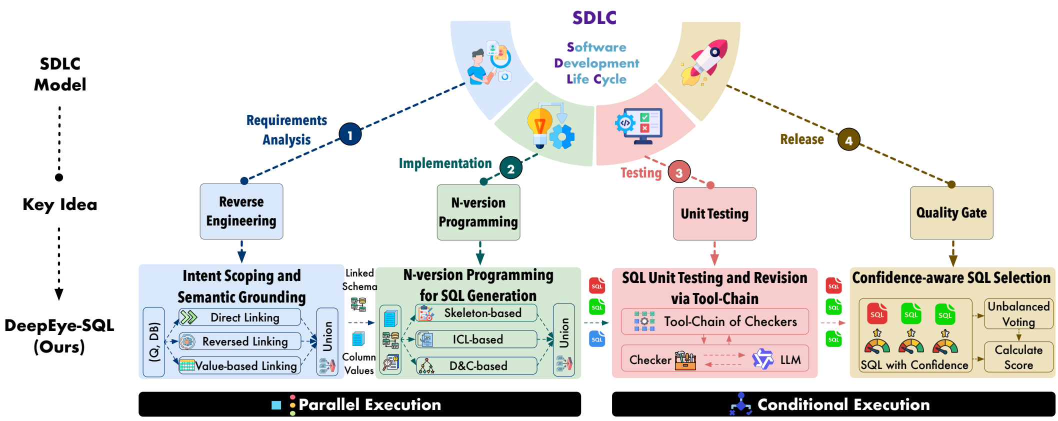


Fig. 1. Key Idea of DeepEye-SQL.

Table 1. Comparison of DeepEye-SQL with representative SOTA Text-to-SQL methods.

Method	Core Paradigm	Verification Mechanism	SQL Selection Strategy	Efficiency & Cost Profile
CHESS [46]	**Linear Inference Workflow** (Single-path, vulnerable to error propagation)	**LLM Self-Refinement** (Probabilistic, hallucination-prone)	**Heuristic Majority Voting** (Standard Self-Consistency)	**High API Cost** (Gemini 1.5 Pro, Proprietary)
Alpha-SQL [18]	**Search-Centric (MCTS)** (Exploration via Monte Carlo Search)	**Basic Syntax Check** (Ensures executability only)	**Heuristic Majority Voting** (Relies on rollout counts)	**High Inference Latency** (MCTS requires massive inferences)
OmniSQL [19]	**Data-Centric SFT** (Synthetic Data Augmentation)	**Basic Syntax Check** (Ensures executability only)	**Heuristic Majority Voting** (Standard Self-Consistency)	**High Training Cost** (Requires Full Fine-Tuning)
DeepEye-SQL (Ours)	**SE-Lifecycle Centric** *(Fault-Tolerant Design)*	**Deterministic Unit Testing** *(Syntax/Logic/Quality Checkers)*	**Confidence-Aware Selection** *(Pairwise Adjudication)*	**Efficient & Training-Free** *(~3B Activated Params)*

example, Alpha-SQL [18] leverages dynamic multi-step reasoning, while XiYan-SQL [24] improves SQL generation and multi-candidate SQL selection through task-specific fine-tuning.

Despite these advances, state-of-the-art performance on the BIRD dataset remains around 70% execution accuracy [21], and reliability further degrades in real-world deployments [6, 17, 58]. This observation indicates that recent advances, while promising, have yet to translate into consistent system-level reliability.

The key limitation lies not in the optimization of individual modules – such as schema linking, reasoning, or post-hoc verification – ***but in the lack of coherent orchestration that enforces correctness across the entire workflow*** [9, 18, 24, 34, 35, 46, 52]. As a result, current Text-to-SQL solutions struggle to: *(i) define what should be built*, which requires precisely determining the semantic scope of the user question and grounding it to the relevant database entities through comprehensive schema linking and value retrieval; *(ii) implement the solution (i.e., SQL generation)*, which involves generating executable SQL queries that faithfully capture the inferred semantics while maintaining diversity and completeness across complex reasoning paths; *(iii) verify its correctness*, which requires systematically validating the structural, logical, and semantic correctness of the generated SQL through interpretable checks [23]; and *(iv) release the generated SQL*, which requires quantifying confidence through multi-source evidence and establishing measurable acceptance criteria for determining whether a generated SQL query is reliable enough for output.

This fundamental gap motivates a paradigm shift: ***Text-to-SQL should be viewed not simply as a language generation task powered by LLMs, but as a software-engineering problem that requires structured orchestration and verifiable correctness.*** From this perspective, generating a correct SQL query resembles developing a small software program: the system must infer user requirements from natural-language questions with respect to the specified database, realize the intended logic through SQL generation, and ensure its correctness and reliability through systematic testing and quality control. Inspired by the Software Development Life Cycle (SDLC) [40] and illustrated in Figure 1, we structure the Text-to-SQL generation workflow as a unified process that integrates requirement analysis from natural-language questions, SQL generation, verification of generated queries, and final release through quality-gated control. However, implementing this idea in practice is non-trivial and poses several **challenges**.

First, Text-to-SQL solutions must *infer* user intent from ambiguous natural language and partially observed database schemas. We term this challenge *ambiguous requirement inference* (challenge **C1**). In the implementation stage, current methods rely on a single reasoning path driven by one model [35] and one prompt configuration [17], *resulting in insufficient fault tolerance* (**C2**). In the verification and validation stage, existing methods depend on probabilistic signals such as self-consistency [18] or partial execution feedback [61] rather than deterministic oracles to assess generated SQL. We denote this challenge as *unreliable verification and validation* (**C3**). In the release stage, current methods lack *calibrated confidence estimation and measurable acceptance criteria* for evaluating when a generated SQL query is reliable enough for output (**C4**).

Our Methodology. To systematically address these challenges, we propose DeepEye-SQL, which reframes Text-to-SQL as a verifiable SDLC-style workflow with four stages (Figure 1). In requirements analysis, we propose Semantic Value Retrieval and a fault-tolerant Robust Schema Linking strategy to build a complete, database-grounded specification (addressing **C1**). In implementation, we adopt N-version programming [4] by executing multiple independent generators with diverse reasoning paradigms in parallel to diversify reasoning under a fixed budget, providing fault tolerance (addressing **C2**). For verification and validation, we replace probabilistic self-judgment with a suite of specialized, deterministic checkers that trigger targeted LLM repair, ensuring verifiable correctness (addressing **C3**). Finally, in release, we introduce Confidence-aware SQL Selection that dynamically adapts the review process based on execution consensus, yielding a calibrated output (addressing **C4**). As shown in Figure 2, this design enables DeepEye-SQL to integrate with diverse LLMs and achieve notable accuracy improvements without any fine-tuning. To explicitly distinguish our contribution from existing approaches, Table 1 contrasts DeepEye-SQL with representative state-of-the-art methods. Unlike linear workflows (e.g., CHESS [46]) or search-centric methods (e.g., Alpha-SQL [18]) that rely heavily on single-path reasoning or probabilistic self-correction, DeepEye-SQL enforces a *deterministic SE-Lifecycle* paradigm. We also incorporate a fault-tolerant design via *N-Version Programming* to mitigate single-point reasoning failures. Furthermore, by replacing stochastic checks with a rigorous *Unit Testing Tool-Chain* and introducing *Confidence-Aware Selection*, we systematically address the inherent unreliability of LLM self-reflection.

Contributions. This paper makes the following contributions:

(1) **A Novel SE-Inspired Framework.** We propose DeepEye-SQL, which reframes Text-to-SQL as a verifiable SDLC workflow. Unlike prior frameworks constrained by *fragile, linear inference chains* where errors propagate unchecked, DeepEye-SQL enforces a *robust engineering lifecycle* (Requirement → Implementation → Testing → Release) to guarantee system-level reliability.

(2) **N-Version Programming for Diversity.** Departing from standard self-consistency methods that rely on stochastic sampling (same prompt, high temperature), we introduce an *N-Version Programming* paradigm. By orchestrating independent generators with distinct reasoning strategies, we achieve true algorithmic fault tolerance and broader coverage of complex queries.

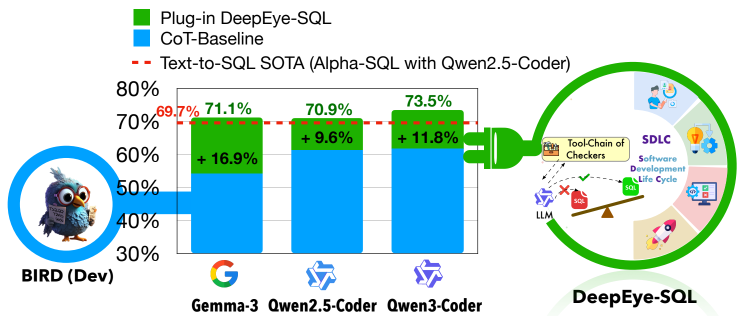


Fig. 2. As a plug-and-play framework, DeepEye-SQL consistently outperforms SOTA baselines using ~30B LLMs.

(3) **Deterministic Verification via Tool-Chain.** To address the confirmation bias inherent in LLM self-correction, we propose a novel *SQL Unit Testing Tool-Chain*. This module replaces probabilistic LLM self-reflection with deterministic, external execution checks (Syntax, Logic, and Data Quality), providing precise feedback that grounds the revision process.

(4) **Extensive Experiments** demonstrate that DeepEye-SQL achieves state-of-the-art performance on challenging benchmarks. Leveraging open-source MoE LLMs with only ~3B activated parameters, DeepEye-SQL attains an execution accuracy of 73.5% on BIRD-Dev, 75.07% on the held-out BIRD-Test set, and 89.8% on Spider-Test, outperforming larger or fine-tuned baselines.

2 DeepEye-SQL Overview

DeepEye-SQL is systematically organized as a multi-stage pipeline inspired by the core principles of the Software Development Life Cycle (SDLC). Rather than reproducing the entire SDLC process literally, it abstracts and automates its essential workflow for Text-to-SQL. Conceptually, DeepEye-SQL serves as a compressed and self-contained microcosm of software development, where the "software" being constructed is a single, correct SQL query. Following this paradigm, the framework comprises four stages (Figure 3), each aligned with a corresponding phase of the SDLC.

Phase-1: Intent Scoping and Semantic Grounding. Mirroring the initial phase of any engineering project—answering "What should be built?"—this stage is dedicated to accurately interpreting user intent and defining the problem's scope. It employs *Semantic Value Retrieval* to ground the query in the database's actual data and *Robust Schema Linking* to identify necessary tables and columns. This linking module uses a fault-tolerant hybrid strategy that integrates complementary linking paradigms to avoid the "*single point of failure*" problem [48], ensuring a comprehensive and fault-tolerant specification.

Phase-2: N-version Programming for SQL Generation. Analogous to the implementation phase, this stage generates SQL queries. To enhance robustness, we employ a strategy inspired by *N-version programming* [4], a fault-tolerance technique where multiple, independent implementations are created for the same problem. Our framework instantiates this by producing a diverse set of SQL candidates in parallel from three distinct generators, each utilizing a unique reasoning paradigm. It is crucial to distinguish this approach from test-time scaling techniques like self-consistency [49], which generate diversity by sampling multiple outputs from a single generator. In contrast, our method achieves a more principled and profound diversity, akin to true N-version programming, as each generator employs a fundamentally different reasoning process. This engineered diversity

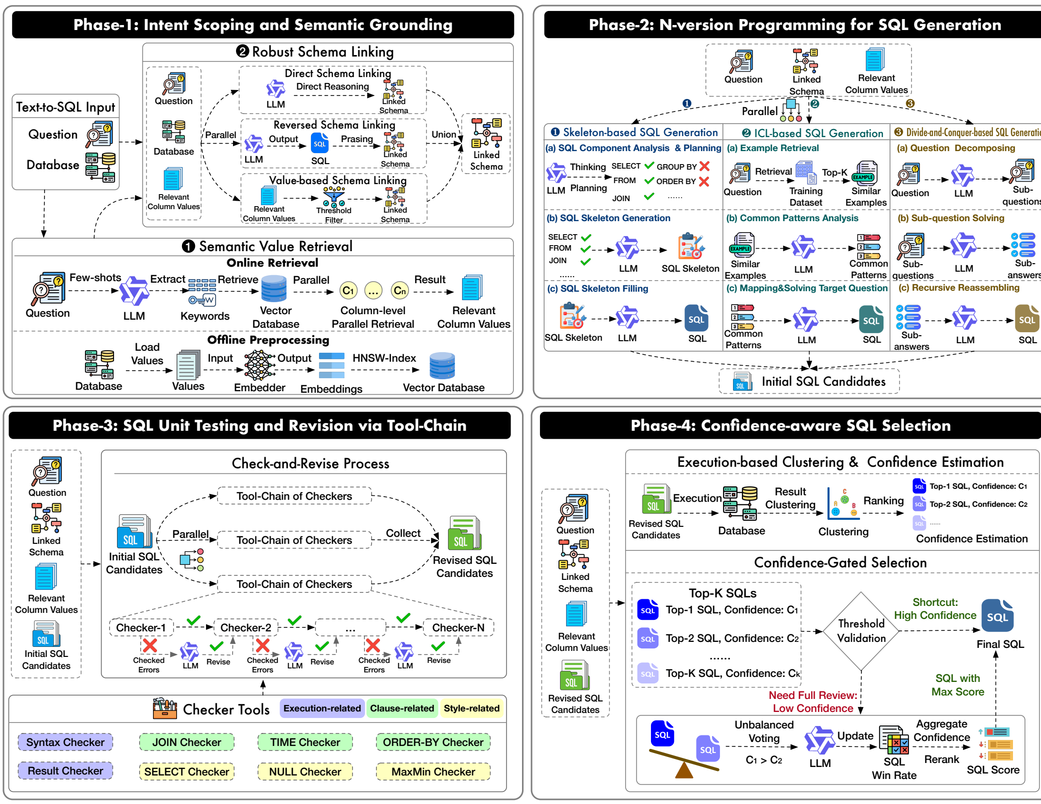


Fig. 3. DeepEye-SQL Overview.

ensures a broader exploration of the solution space, significantly increasing the probability of producing at least one correct query, especially for complex scenarios where a single reasoning path might fail.

Phase-3: SQL Unit Testing and Revision via Tool-Chain. This phase embodies the critical software engineering principle of *Unit Testing* [39]. Its purpose is to systematically verify the correctness of each generated SQL candidate and revise any found defects. To overcome the known unreliability of LLM self-correction [60], our framework externalizes this process, emulating a rigorous, automated testing loop. Each SQL candidate is passed through a *Tool-Chain of Checkers*—a suite of specialized, deterministic tools where each checker acts as a test case for a specific unit of functionality. These checkers systematically validate structural and semantic constraints, and if a flaw is detected, provide *an explicit and actionable directive* to the LLM for a targeted revision, mirroring a formal bug report and fix cycle.

Phase-4: Confidence-aware SQL Selection. The final phase corresponds to the release stage, governed by a *Quality Gate*. Instead of simply choosing the most common answer, this stage arbitrates which candidate is reliable enough to be "released". Our *Confidence-aware SQL Selection* mechanism performs this task. It estimates the reliability of candidates based on execution consensus and evaluates this against a predefined acceptance criterion. This validation acts as a gateway, determining whether to release the result directly or trigger a deeper, comparative review to ensure the most reliable query is selected.

Algorithm 1 Online Semantic Value Retrieval

Input: User Question Q, Set of Column Vector Indices $\{\mathcal{I}_j\}_{j=1}^{M}$, Top-K parameter K
Output: Retrieved Values Map $\mathcal{M}_{retrieved}$: $C_j \to \mathcal{V}_j$

// Step 1: Extract Keywords from the user question
$\mathcal{K} \leftarrow \texttt{LLM} - \texttt{ExtractKeywords}(Q)$
// Step 2: Retrieve values in parallel for each indexed column
Initialize $\mathcal{M}_{retrieved} \leftarrow \emptyset$
for all index $\mathcal{I}_j$ for column C_j **in parallel do**
  $\mathcal{V}_{candidates} \leftarrow \emptyset$
  for all keyword $k_i \in \mathcal{K}$ **do**
    $\mathbf{e}_{k_i} \leftarrow \texttt{Embed}(k_i)$
    $\mathcal{V}_{partial} \leftarrow \texttt{SearchIndex}(\mathcal{I}_j, \mathbf{e}_{k_i}, K)$
    $\mathcal{V}_{candidates} \leftarrow \mathcal{V}_{candidates} \cup \mathcal{V}_{partial}$
  end for
  $\mathcal{V}_{sorted} \leftarrow \texttt{SortBySimilarity}(\mathcal{V}_{candidates})$ *// Aggregate and select top-K unique values*
  $\mathcal{V}_j \leftarrow \texttt{GetUniqueTopK}(\mathcal{V}_{sorted}, K)$
  $\mathcal{M}_{retrieved}[C_j] \leftarrow \mathcal{V}_j$
end for
return $\mathcal{M}_{retrieved}$

3 The Design Details of DeepEye-SQL

3.1 Semantic Value Retrieval

A key Text-to-SQL challenge is the grounding problem: LLMs lack awareness of specific database values, often generating SQL with hallucinated or mismatched filter values (*e.g.*, using `country = 'USA'` when the database stores `'United States'`). This issue is particularly prevalent for high-cardinality, free-form text columns. Addressing this mirrors the software engineering principle of *dependency resolution* [33], which requires a system to be aware of valid data constants it can operate on. To mitigate this, DeepEye-SQL incorporates a *Semantic Value Retrieval* module that proactively supplies the LLM with a contextually relevant subset of database values, anchoring the generation process to the ground-truth data. Our value retrieval process is divided into two distinct stages: an efficient, one-time offline preprocessing stage for index construction, and a rapid online retrieval stage executed at query time.

Offline Preprocessing. The primary objective of the offline phase is to preprocess and index database values to enable efficient semantic search. This process is executed once for any new database and involves three steps.

Selective Value Extraction. Instead of a brute-force approach that indexes every value—which would be computationally prohibitive and introduce significant noise—we perform selective extraction. We specifically target columns of type `TEXT`, as they are the primary source of ambiguity and value-related hallucinations. To further refine this process, we apply heuristics to exclude columns that, despite being text-based, are unlikely to be used in semantic comparisons, such as columns containing UUIDs or exclusively numerical identifiers. This strategic selection minimizes the indexing overhead while enhancing the semantic relevance of the retrieved values.

Value Embedding. For each selected column C_j, we extract its unique values. Each distinct value v is then encoded into a high-dimensional vector representation $\mathbf{e}_v$ using a pretrained sentence embedding model, specifically `Qwen3-Embedding-0.6B` [62]. This embedding transforms discrete text

Algorithm 2 Robust Schema Linking

Input: User Question Q, Database Schema $\mathcal{D}$, Retrieved Values Map $\mathcal{M}_{retrieved}$, Threshold θ_{val}
Output: Final Linked Schema $\mathcal{D}_{linked}$

// Step 1: Execute the three linking strategies in parallel
begin parallel
  $\mathcal{D}_{direct} \leftarrow$ `LLM_DirectLink`$(Q, \mathcal{D}, \mathcal{M}_{retrieved})$
  $\mathcal{S}' \leftarrow$ `LLM_GenerateSQL`$(Q, \mathcal{D}, \mathcal{M}_{retrieved})$
  $\mathcal{D}_{reversed} \leftarrow$ `ParseSchema`$(\mathcal{S}')$
  $\mathcal{D}_{value} \leftarrow$ `FindValueBasedSchema`$(\mathcal{M}_{retrieved}, \theta_{val})$
end parallel
// Step 2: Aggregate the results by taking the union
$\mathcal{D}_{union} \leftarrow \mathcal{D}_{direct} \cup \mathcal{D}_{reversed} \cup \mathcal{D}_{value}$
// Step 3: Enforce relational closure to ensure schema connectivity
$\mathcal{D}_{linked} \leftarrow$ `EnforceClosure`$(\mathcal{D}_{union}, \mathcal{D})$
return $\mathcal{D}_{linked}$

strings into continuous semantic vectors, where values with similar meanings are located closer to each other in the vector space.

Vector Indexing. To facilitate fast similarity search, the generated value embeddings for each column are used to build a vector index. We employ Chroma [5] with the Hierarchical Navigable Small World (HNSW) algorithm [32] for this purpose. HNSW is highly efficient for approximate nearest neighbor (ANN) search, making it ideal for real-time applications. The result is a set of persistent, per-column vector indices $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_M\}$, where each $\mathcal{I}_j$ is an indexed collection of all value embeddings for a column C_j.

Online Retrieval. During the online phase, when a user question Q is received, the system retrieves the most relevant values for that question from the pre-built indices. This process is detailed in Algorithm 1.

Keyword Extraction. First, we leverage the LLM to identify potential entities and filter values within the user's question Q. Using a few-shot prompting strategy, we instruct the LLM to extract a set of key terms $\mathcal{K} = \{k_1, k_2, ..., k_N\}$ that are likely to appear in a `WHERE` clause. For example, from the question "Show me all papers by authors from France", the LLM would extract keywords like "France".

Parallel Multi-Column Retrieval. With the extracted keywords, we perform a parallel search across all indexed columns. For each indexed column C_j, we perform the following steps:

(1) For every keyword $k_i \in \mathcal{K}$, we generate its embedding $\mathbf{e}_{k_i}$ using the same model from the offline phase.
(2) We query the corresponding column index $\mathcal{I}_j$ with $\mathbf{e}_{k_i}$ to retrieve the top-K most similar values along with their similarity scores (*e.g.*, cosine similarity).
(3) After querying for all N keywords, we obtain $N \times K$ candidate values for column C_j. We aggregate these candidates, sort them globally by their similarity scores in descending order, and select the top-K unique values. This yields the final retrieved value set $\mathcal{V}_j$ for column C_j.

3.2 Robust Schema Linking

Schema linking, the task of identifying the correct subset of tables and columns relevant to a user's question, is a cornerstone of any Text-to-SQL system [22]. Existing methods [17, 18, 46] often treat this as a direct mapping task, which can be brittle when faced with complex schemas or ambiguous questions. An error at this stage is catastrophic, as an incomplete or incorrect schema makes generating a correct SQL query nearly impossible. This critical dependency is analogous to the role of a *formal specification* [10] in software development; without an accurate specification, the final product is destined to fail. To address this, we introduce a *Robust Schema Linking* module that, inspired by the principle of fault tolerance, combines multiple, diverse strategies to ensure the most accurate and complete schema is identified. Our overall process is detailed in Algorithm 2.

Our approach is a hybrid model that integrates three complementary linking techniques: (1) **Direct Schema Linking**, (2) **Reversed Schema Linking**, and (3) **Value-based Schema Linking**.

Direct Schema Linking. This method represents the most conventional approach, directly tasking the LLM with acting as a schema analysis agent. Given the user's question Q, the full database schema $\mathcal{D}$, and the retrieved relevant values $\mathcal{M}_{retrieved}$, we prompt the LLM to explicitly list all relevant schema components. This process can be formalized as:

$$\mathcal{D}_{direct} = \text{LLM}_{\text{DirectLink}}(Q, \mathcal{D}, \mathcal{M}_{retrieved}) \tag{1}$$

While effective for unambiguous queries, its performance can degrade in complex scenarios, making it an unreliable standalone solution.

Reversed Schema Linking. Inspired by the software engineering practice of *reverse engineering* [38], this technique reimagines the schema linking process. Instead of asking the LLM to first identify the schema, we prompt it to generate a draft SQL query directly, providing it with the full context including the relevant values. This approach is more effective because the task of generating SQL code aligns more closely with an LLM's pre-training on vast code corpora [44]. We then use a static parser to extract all schema components from the generated query S'. Formally, the process is:

$$\mathcal{D}_{reversed} = \text{ParseSchema}(\text{LLM}_{\text{GenerateSQL}}(Q, \mathcal{D}, \mathcal{M}_{retrieved})) \tag{2}$$

This "answer-first" approach allows the LLM to implicitly perform schema linking, often revealing components for complex joins or subqueries that a direct analysis might miss.

Value-based Schema Linking. This technique provides an empirical, data-driven check to complement the model-driven approaches. It operates on the principle that if a column contains values highly similar to keywords in the question, that column is likely relevant. This method leverages the retrieved values $\mathcal{M}_{retrieved}$ from the previous module. A column C_j is selected if any of its retrieved values has a similarity score with any question keyword $k \in \mathcal{K}$ that exceeds a high-confidence threshold θ_{val}. This can be expressed as:

$$\begin{aligned}\mathcal{D}_{value} = \{C_j \in \mathcal{D} \mid & \exists v \in \mathcal{M}_{retrieved}[C_j], \exists k \in \mathcal{K} \\ & \text{s.t. } \text{Sim}(v, k) > \theta_{val}\}\end{aligned} \tag{3}$$

This bottom-up method excels at resolving schema ambiguity. In cases where multiple columns are plausible candidates, it precisely identifies the correct one by grounding the selection in concrete data values instead of potentially misleading column names.

Schema Union and Closure. The final step aggregates the results and ensures the relational integrity of the linked schema. First, we take the union of the schemas identified by all three methods:

$$\mathcal{D}_{union} = \mathcal{D}_{direct} \cup \mathcal{D}_{reversed} \cup \mathcal{D}_{value} \tag{4}$$

However, since $\mathcal{D}_{union}$ is aggregated from independent strategies, it acts effectively as a collection of isolated schema nodes (tables). Without explicit edges (Foreign Keys) to connect them, the resulting schema graph is often disconnected, rendering valid JOIN operations impossible. To

Algorithm 3 N-version Programming for SQL Generation

Input: User Question Q, Linked Schema $\mathcal{D}_{linked}$, Retrieved Values Map $\mathcal{M}_{retrieved}$
Output: Set of Initial SQL Candidates $C_{initial}$

// Step 1: Retrieve few-shot examples for the ICL generator
$\mathcal{E}_{few-shot} \leftarrow$ `RetrieveSimilarExamples`(Q)
// Step 2: Execute the three SQL generators in parallel
begin parallel
  $\mathcal{S}_{skel} \leftarrow$ `LLM_Skel`$(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved})$
  $\mathcal{S}_{icl} \leftarrow$ `LLM_ICL`$(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}, \mathcal{E}_{few-shot})$
  $\mathcal{S}_{d\&c} \leftarrow$ `LLM_D&C`$(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved})$
end parallel
// Step 3: Collect all generated SQLs into a candidate set
$C_{initial} \leftarrow \{\mathcal{S}_{skel}, \mathcal{S}_{icl}, \mathcal{S}_{d\&c}\}$ *// and other samples if N > 1 per generator*
return $C_{initial}$

resolve this, we enforce relational closure as a mandatory connectivity safeguard. We parse all foreign key relationships in the database schema $\mathcal{D}$. For any pair of tables (T_i, T_j) present in $\mathcal{D}_{union}$, we automatically add the corresponding primary and foreign key columns that link them. This step transforms the set of isolated tables into a connected graph, providing a solid topological foundation for the subsequent SQL generation phase. This final, closed schema is defined as:

$$\mathcal{D}_{linked} = \text{EnforceClosure}(\mathcal{D}_{union}, \mathcal{D}) \tag{5}$$

3.3 N-version Programming for SQL Generation

Upon establishing the query's specification in the *Intent Scoping and Semantic Grounding* phase, the framework proceeds to the implementation stage: SQL generation. A single generation strategy, however, often struggles with the diversity of user queries [34]; a method effective for simple lookups may fail on complex analytical questions [35]. To address this, DeepEye-SQL instantiates a fault-tolerant strategy inspired by the software engineering principle of *N-version programming* [4]. Instead of relying on a single, monolithic generator, we deploy three distinct and independent SQL generators that run in parallel, each employing a different methodology. This engineered diversity significantly increases the likelihood of producing at least one correct candidate, enhancing the system's overall robustness. The entire workflow is detailed in Algorithm 3.

The inputs to this phase are the user question Q, the linked schema $\mathcal{D}_{linked}$, and the retrieved values $\mathcal{M}_{retrieved}$. The three generators operate on this common set of inputs to produce a unified pool of initial SQL candidates $C_{initial}$.

Skeleton-based SQL Generation. This generator is modeled after the *top-down design* principle [47], where a high-level plan is formulated before implementation details are filled in. This guides the LLM to think systematically, reducing structural errors. It involves three conceptual steps: component analysis, skeleton generation, and slot-filling. The entire process is encapsulated in a single call to the LLM, which is instructed to follow this reasoning path. We can formalize this as:

$$S_{skel} = \text{LLM}_{\text{Skel}}(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}) \tag{6}$$

ICL-based SQL Generation. This generator leverages in-context learning (ICL), analogous to *case-based reasoning* [14]. By providing the LLM with relevant examples, we ground its generation in proven patterns. To eliminate domain-specific bias, we adopt a masked retrieval strategy following DAIL-SQL [9]. Specifically, we mask table and column names in the user question with a special

Algorithm 4 SQL Unit Testing and Revision via Tool-Chain

Input: A single SQL candidate $\mathcal{S}_{cand}$, Context $(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved})$, Tool-Chain $\mathbb{C} = \{C_1, ..., C_N\}$
Output: Revised SQL query $\mathcal{S}_{revised}$

$\mathcal{S}_{current} \leftarrow \mathcal{S}_{cand}$
for all checker $C_j \in \mathbb{C}$ **do**
  $is\_valid, d_{err} \leftarrow \mathtt{C_j}(\mathcal{S}_{current})$
  if not is_valid **then**
  // Error found, trigger revision and update the current SQL
    $\mathcal{S}_{current} \leftarrow \mathtt{LLM}_{\text{Revise}}(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}, \mathcal{S}_{current}, d_{err})$
  end if
end for
// All checkers in the chain have been processed
return $\mathcal{S}_{current}$

token (*i.e.,* `<MASK>`) to focus solely on structural logic. We then encode these masked questions using the `all-MiniLM-L6-v2` [37] model and retrieve the top-K most similar examples from the training set of the corresponding benchmark (*e.g.,* BIRD-Train) to serve as few-shot demonstrations. This is formalized as:

$$S_{icl} = \text{LLM}_{\text{ICL}}(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}, \mathcal{E}_{few-shot}) \tag{7}$$

where $\mathcal{E}_{few-shot}$ represents the set of retrieved few-shot examples.

Divide-and-Conquer-based SQL Generation. For highly complex questions requiring nested logic, this generator implements the classic *Divide and Conquer* paradigm. It breaks a large problem into smaller, manageable sub-problems that are solved recursively and then reassembled. This involves decomposing the question, solving each sub-question, and synthesizing the results into a single query. The process is formalized as:

$$S_{d\&c} = \text{LLM}_{\text{D\&C}}(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}) \tag{8}$$

3.4 SQL Unit Testing and Revision via Tool-Chain

LLM-generated SQL often contains critical errors (*e.g.,* incorrect `JOIN`s), yet LLMs struggle to self-correct due to inherent confirmation bias [23, 36, 60]. Analogous to the principle that code requires independent validation, we introduce *SQL Unit Testing and Revision via Tool-Chain.* Drawing on *Unit Testing* [39], we treat functional SQL components (*e.g.,* `JOIN` logic, syntax) as testable "units". The *Tool-Chain* externalizes verification through a deterministic chain of specialized checkers, functioning as an automated test suite to systematically detect errors and guide targeted revisions.

Tool-Chain of Checkers.

Our approach centers on a suite of specialized, deterministic programs we term "Checker Tools". These checkers are not applied randomly; they are orchestrated into a sequential *"Syntax → Logic → Quality" cascade.* This order is critical for a "fail-fast" verification: it ensures that fatal execution errors are resolved first before the system attempts to diagnose more subtle logical or stylistic discrepancies. The complete, ordered sequence is detailed in Table 2. The chain begins with the most fundamental validation: the *Syntax Checker* ensures the query is syntactically valid and executable. Notably, this stage also acts as an implicit guard for aggregation errors (e.g., missing `GROUP BY`), as these typically trigger execution failures that are immediately captured and repaired. Once executable, the chain proceeds to *Logical Verification.* The *JOIN Checker* flags non-standard conditions, while the *ORDER-BY Checker* validates sorting logic and corrects invalid patterns (e.g., misaligned `LIMIT` clauses). Finally, the chain addresses *Quality and Robustness* issues. The *SELECT*

Table 2. Tool-chain of specialized checkers organized by verification stage.

Checker	Error Detection	Example Cases
	Stage 1: Syntax & Execution (Fail-Fast)	
Syntax Checker	Syntax validity and execution errors	`SELECT col, COUNT(*)` (missing `GROUP BY`); `SELECT * FORM table` (typo)
	Stage 2: Logical Verification	
JOIN Checker	Non-standard or complex joins	`ON T1.id = T2.id OR ...;ON col IN (SELECT ...)`
ORDER-BY Checker	Logical conflicts in sorting and limits	`ORDER BY COUNT(*) LIMIT 1` (validate aggregation-based sorting)
Time Checker	Invalid temporal function usage	`STRFTIME` vs. `DATETIME`; invalid date-format comparisons
	Stage 3: Quality & Robustness	
SELECT Checker	Ambiguity removal	`SELECT *` → `SELECT col_a, col_b`
MaxMin Checker	Efficiency optimization	`WHERE col = (SELECT MAX(...))` → `ORDER BY col DESC LIMIT 1`
NULL Checker	Missing NULL guards	Add `WHERE col IS NOT NULL` to sorting keys
Result Checker	Empty-result verification	Queries returning zero rows (triggering re-evaluation of constraints)

Checker replaces ambiguous wildcards like `SELECT *` with specific column names. Concurrently, the *NULL Checker* and *Result Checker* inject guards for potential `NULL` issues and flag queries that produce empty results, ensuring the final output is not only correct but also practically useful.

The Sequential Check-and-Revise Process. Our framework implements an efficient, single-pass "check-and-revise" process, detailed in Algorithm 4. Each initial SQL candidate, $\mathcal{S} \in C_{initial}$, is passed through the *Tool-Chain of Checkers* exactly once. The process for a single candidate $\mathcal{S}_{cand}$ is as follows:

(1) The candidate $\mathcal{S}_{cand}$ is sequentially evaluated by each checker in the tool-chain, starting with the first.
(2) If a checker detects an error, the process is momentarily paused. The checker generates a specific error report and an actionable *correction directive*, d_{err}. For example, if the `NULL Checker` finds that a column in an `ORDER BY` clause could contain `NULL` values, the directive would be a clear instruction such as: "The ordering column `[column_name]` may contain NULLs. Add a `WHERE [column_name] IS NOT NULL` condition to ensure correct sorting."
(3) The LLM is then invoked in a special "revision mode". It receives the original context, the faulty SQL $\mathcal{S}_{cand}$, and the explicit directive d_{err} from the checker. The LLM's task is not to find the error, but to fix it based on the directive. It is formalized as:

$$\mathcal{S}_{revised} = \text{LLM}_{\text{Revise}}(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved}, \mathcal{S}_{cand}, d_{err}) \tag{9}$$

(4) The newly revised query, $\mathcal{S}_{revised}$, replaces $\mathcal{S}_{cand}$, and the evaluation continues with the next checker in the chain.
(5) The process terminates once the query has been evaluated by all checkers in the chain.

Algorithm 5 Confidence-aware SQL Selection

Input: Revised SQL Candidates $\mathcal{C}_{revised}$, Context $(Q, \mathcal{D}_{linked}, \mathcal{M}_{retrieved})$, Threshold θ_{conf}
Output: The Final SQL Query $\mathcal{S}_{final}$

// Step 1: Execute all candidates and cluster the results
$\mathcal{R} \leftarrow \texttt{ExecuteAll}(\mathcal{C}_{revised})$
$\{\text{Cluster}_1, ..., \text{Cluster}_M\}, \{\mathcal{S}_1, ..., \mathcal{S}_M\} \leftarrow \texttt{ClusterAndRank}(\mathcal{R})$
// Step 2: Calculate confidence for the top-ranked candidate
$Conf(\mathcal{S}_1) \leftarrow \frac{|\text{Cluster}_1|}{|\mathcal{C}_{revised}|}$
// Step 3: Confidence-Gated Selection Path
if $Conf(\mathcal{S}_1) > \theta_{conf}$ **then**
    $\mathcal{S}_{final} \leftarrow \mathcal{S}_1$ *// High-Confidence Shortcut*
else
    Let $\{\mathcal{S}_1, ..., \mathcal{S}_K\}$ be the top-K candidates *// Low-Confidence Full Review*
    for all candidate $\mathcal{S}_i$ in $\{\mathcal{S}_1, ..., \mathcal{S}_K\}$ **do**
        $Conf(\mathcal{S}_i) \leftarrow \frac{|\text{Cluster}_i|}{|\mathcal{C}_{revised}|}$
        $WinRate(\mathcal{S}_i) \leftarrow \texttt{LLM-PairwiseVoting}(\{\mathcal{S}_1, ..., \mathcal{S}_K\})$ *// Using Eq. 11*
        $Score(\mathcal{S}_i) \leftarrow Conf(\mathcal{S}_i) \times WinRate(\mathcal{S}_i)$
    end for
    $\mathcal{S}_{final} \leftarrow \arg\max_{\mathcal{S}_i} Score(\mathcal{S}_i)$
end if
return $\mathcal{S}_{final}$

This external, tool-guided debugging process is significantly more reliable than unconstrained LLM self-correction. The final output of this phase is a set of revised SQL candidates, $\mathcal{C}_{revised}$, which have been rigorously vetted and have a substantially higher probability of being correct.

3.5 Confidence-aware SQL Selection

The preceding phases of our framework produce a set of high-quality, revised SQL candidates, $\mathcal{C}_{revised}$. However, these candidates may not be identical and could yield different execution results. The most common approach for selecting a final query is *self-consistency* [9, 18, 52], where all candidates are executed, and the query corresponding to the most frequent result is chosen. This method, while a strong baseline, has a critical flaw: *the most popular answer is not always the correct one, especially for complex problems where multiple generation paths might converge on the same plausible but incorrect logic.* This final challenge mirrors the release stage in a software lifecycle, which is governed by a *Quality Gate* [41]. A quality gate's purpose is not merely to accept the most-voted-for version, but to enforce a set of objective quality criteria before a product is released. Similarly, our *Confidence-aware SQL Selection* phase acts as this quality gate for the generated SQL. It overcomes the flaws of simple majority voting by using the initial vote's confidence as a quality metric to guide a more reliable, adaptive selection process.

Execution-based Clustering and Confidence Estimation. The process begins by executing every revised SQL candidate $\mathcal{S} \in \mathcal{C}_{revised}$ on the database. The resulting datasets are then clustered, such that queries producing identical results are grouped together. These clusters are ranked based on their size (*i.e.,* the number of SQL queries they contain). For each of the top-K candidates $\{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_K\}$, representing the top-$K$ largest clusters, we calculate an execution-based confidence score. The confidence of a candidate $\mathcal{S}_i$ is the proportion of total queries that belong to its cluster:

$$Conf(\mathcal{S}_i) = \frac{|\text{Cluster}_i|}{|\mathcal{C}_{revised}|} \tag{10}$$

This score, particularly $Conf(\mathcal{S}_1)$, serves as a strong indicator of the query's likely correctness, as shown by the high correlation in Figure 6.

Confidence-Gated Selection. Based on the confidence of the top-ranked candidate $\mathcal{S}_1$, our framework follows one of two distinct paths, as detailed in Algorithm 5.

High-Confidence Shortcut. If the confidence score $Conf(\mathcal{S}_1)$ exceeds a predefined high-confidence threshold θ_{conf}, we conclude that there is overwhelming agreement among the generated candidates. In this scenario, we directly select $\mathcal{S}_1$ as the final query. This shortcut avoids unnecessary and costly LLM invocations for cases where the answer is already clear, providing a practical trade-off between accuracy and efficiency.

Low-Confidence Full Review. If $Conf(\mathcal{S}_1) < \theta_{conf}$, it signifies substantial ambiguity or disagreement among the candidates, making the top choice unreliable. It then triggers a full review pipeline.

(1) **Execution-Guided Prompting.** Instead of a neutral comparison, we construct an asymmetric prompt guided by execution consensus. We identify the top-K execution clusters and randomly select one representative SQL from each cluster to form the candidate set. The prompt explicitly injects the execution confidence as context (e.g., informing the LLM that Candidate A has higher execution confidence than Candidate B). We instruct the LLM to treat this high-confidence consensus as a prior, trusting it unless specific *logical evidence*—such as a contradiction between the generated SQL and the user's natural language requirement—demonstrably proves it factually incorrect.

(2) **Perform Pairwise Adjudication.** Next, the LLM performs pairwise comparisons for the top-K candidates. To ensure reliability and mitigate the inherent stochasticity of LLM outputs, we employ a *self-consistency* mechanism. For each pair $(\mathcal{S}_i, \mathcal{S}_j)$, we sample three independent judgments and define the final stable vote, $V(\mathcal{S}_i, \mathcal{S}_j)$, as the majority outcome. The vote $V(\mathcal{S}_i, \mathcal{S}_j)$ yields 1 if $\mathcal{S}_i$ is superior and 0 if $\mathcal{S}_j$ is superior. From these consistent pairwise results, we compute an aggregate win rate for each candidate $\mathcal{S}_i$ as its average score against all other competitors in the top-K set:

$$WinRate(\mathcal{S}_i) = \frac{1}{K-1} \sum_{j=1, j\neq i}^{K} V(\mathcal{S}_i, \mathcal{S}_j) \tag{11}$$

(3) **Calculate Final Score.** Finally, the decision is based on a confidence-aware score that combines the prior execution-based confidence with the LLM-adjudicated win rate:

$$Score(\mathcal{S}_i) = Conf(\mathcal{S}_i) \times WinRate(\mathcal{S}_i) \tag{12}$$

The query with the highest overall score, $\arg\max_{\mathcal{S}_i} Score(\mathcal{S}_i)$, is selected as the final SQL, $\mathcal{S}_{final}$.

3.6 Workflow Optimization

While DeepEye-SQL's multi-stage architecture ensures robustness, we incorporate three key optimizations to maintain efficiency and computational costs, preventing prohibitive latency or expense.

Efficient Prompting Strategy. Instead of sequential, costly LLM API calls for multi-step reasoning (*e.g.,* planning), we use a single, sophisticated prompt for each generator. This prompt includes a "chain-of-thought" instruction, directing the LLM to perform the entire logical sequence internally and output only the final SQL in one call. This preserves structured reasoning benefits while eliminating the associated latency and token overhead.

Parallel Execution. To reduce end-to-end latency, our framework heavily parallelizes independent tasks. Critical components are executed concurrently, including: multi-column value retrieval, all three schema linking strategies, the N-version SQL generators, and the parallel revision of each SQL candidate.

Conditional Execution. To minimize unnecessary LLM invocations, DeepEye-SQL employs conditional execution at two critical stages. First, during *SQL Unit Testing and Revision*, the LLM is only called for revision if a checker tool detects an error. Second (Section 3.5), LLM-based pairwise adjudication is only triggered in low-confidence scenarios.

4 Experiments

This section evaluates the performance and reliability of DeepEye-SQL. We first detail the experimental setup, followed by a comparative analysis against state-of-the-art methods. Finally, we conduct ablation studies and efficiency analyses to validate the contribution of individual components.

4.1 Experimental Setup

Datasets. We evaluate the performance of DeepEye-SQL on three challenging Text-to-SQL benchmarks: **BIRD** [21], **Spider** [57], and the newly released **Spider 2.0** [16]. BIRD is a large-scale benchmark designed to mirror complex, real-world scenarios. It contains 12,751 unique question-SQL pairs across 95 databases from over 37 professional domains. Its databases are notably large and feature messy data and intricate schemas, making it a difficult test for grounding and robustness. Spider is a foundational large-scale, cross-domain benchmark in the field. It consists of 10,181 questions and 5,693 unique, complex SQL queries across 200 databases covering 138 different domains. **Spider 2.0** represents a significant leap towards enterprise-grade challenges. Unlike its predecessor, it introduces complex schema and diverse dialects (e.g., BigQuery, Snowflake) to simulate real-world data warehousing environments. It is divided into two subsets: *Lite* and *Snow*. Following prior works [3, 8, 24, 34], we use the development set of BIRD (BIRD-Dev), the test set of Spider (Spider-Test), and both the *Snow* and *Lite* subsets of Spider 2.0 for our main evaluation.

Evaluation Metrics. Following prior works [3, 24, 34], our primary evaluation metric is **Execution Accuracy (EX)**. For BIRD and Spider, a generated SQL is considered correct if its execution result is strictly equivalent to the ground truth. For Spider 2.0, we adhere to its official evaluation protocol [16], which employs a relaxed comparison standard that permits redundant columns in the result set, acknowledging the practical needs of data warehousing environments. To measure the potential of our N-version Programming for SQL Generation module, we report the **Upper-bound EX**, which is the execution accuracy an oracle would achieve by always selecting the correct SQL from the generated candidates [34]. For evaluating the Robust Schema Linking module, we use **Table/Column Recall**, the proportion of ground-truth tables and columns correctly identified. Finally, to assess practical efficiency, we measure **Token Cost**, corresponding to the number of tokens processed by LLMs.

Baselines. We compare DeepEye-SQL against SOTA baselines from two paradigms (Table 3). The first is **fine-tuning-based methods**, which are trained on in-domain data, including strong competitors like XiYan-SQL [24], CHASE-SQL [34], and OmniSQL [19]. The second is **prompting-based methods**, which, like DeepEye-SQL, are training-free. This group includes methods like Alpha-SQL [18], RSL-SQL [3], and CHESS [46]. Furthermore, for the Spider 2.0 evaluation, we compare against specialized agentic frameworks designed for enterprise-grade complexity, including Spider-Agent [16], ReFoRCE [8] and LinkAlign [50].

Table 3. Performance comparison on BIRD-Dev and Spider-Test datasets.

Methods	Model	# Parameters	BIRD-EX (%)	Spider-EX (%)
		Fine-tuning-based Baselines		
SENSE [56]	CodeLLaMA-13B	~13B	55.5	86.6
SFT CodeS [20]	CodeS-15B	~15B	58.5	-
Distillery [31]	GPT-4o	>200B	67.2	-
XiYanSQL-QwenCoder [24]	Qwen2.5-Coder-32B-Instruct	~32B	67.1	88.4
BASE-SQL [43]	Qwen2.5-Coder-32B-Instruct	~32B	67.5	88.9
OmniSQL [19]	Qwen2.5-Coder-32B-Instruct	~32B	67.0	89.8
CSC-SQL [42]	XiYanSQL-QwenCoder-32B	~32B	71.3	-
CHASE-SQL [34]	Gemini-1.5-Pro + Gemini-1.5-Flash	>200B	73.0	87.6
XiYan-SQL [24]	GPT-4o + Qwen2.5-Coder-32B-Instruct	>200B	73.3	89.7
		Prompting-based Baselines		
DIN-SQL [35]	GPT-4	>175B	50.7	85.3
DAIL-SQL [9]	GPT-4	>175B	55.9	86.6
SuperSQL [17]	GPT-4	>175B	58.5	-
RSL-SQL [3]	GPT-4o	>200B	67.2	87.9
CHESS (IR,CG,UT) [46]	Gemini-1.5-Pro	>200B	68.3	-
OpenSearch-SQL (v2) [52]	GPT-4o	>200B	69.3	87.1
Alpha-SQL [18]	Qwen2.5-Coder-32B-Instruct	~32B	69.7	-
		Our DeepEye-SQL (Prompting-based)		
DeepEye-SQL	**Gemma3-27B-Instruct**	~27B	**71.1**	**88.9**
DeepEye-SQL	**Qwen2.5-Coder-32B-Instruct**	~32B	**70.9**	**88.7**
DeepEye-SQL	**Qwen3-Coder-30B-A3B-Instruct**	~30B	**73.5**	**89.8**

Implementation Details. All experiments are conducted on an Ubuntu 22.04.3 LTS server with 512GB of RAM and dual 40-core Intel(R) Xeon(R) Platinum 8383C CPUs. We deploy all open-source LLMs locally using the vllm [15] framework on 8 NVIDIA A100 GPUs, each with 80GB of memory, and accelerate inference using a tensor parallelism of 8. To validate the robustness and generalizability of our framework, we integrated DeepEye-SQL with three distinct models from two leading series: `Gemma3-27B-Instruct` [13], `Qwen2.5-Coder-32B-Instruct` [11], and `Qwen3-Coder-30B-A3B-Instruct` [55]. Additionally, for the Spider 2.0 benchmark, which requires extreme reasoning capabilities to navigate complex enterprise schemas, we adopted the `DeepSeek-R1` model [7], consistent with the configuration of comparable baselines. Unless otherwise specified, the pipeline is configured as follows. For Semantic Value Retrieval, where we retrieve up to the top-5 most similar values for each TEXT-type column, we use the `Qwen3-Embedding-0.6B` [62] model, and the similarity threshold (θ_{val}) for Value-based Schema Linking is set to 0.98. The draft SQL for Reversed Schema Linking is produced by the ICL-based generator. For each LLM sub-task (*e.g.,* Direct Schema Linking), we set the sampling budget to 8 with a sampling temperature of 0.7. Finally, in the Confidence-aware SQL Selection phase, which adjudicates between the top-2 ranked queries in low-confidence scenarios, the confidence shortcut threshold (θ_{conf}) is set to 0.6.

4.2 Overall Performance

RQ1: How does DeepEye-SQL perform against existing state-of-the-art methods on challenging Text-to-SQL benchmarks?

To answer this, we present the main performance of DeepEye-SQL on the BIRD-Dev, Spider-Test, and Spider 2.0 datasets in Table 3 and Table 4, comparing it against a wide range of state-of-the-art fine-tuning, prompting-based, and agentic methods. The results clearly demonstrate

Table 4. Execution Accuracy (%) on Spider 2.0 Benchmark. All methods utilize DeepSeek-R1 as the backend LLM to ensure a fair comparison.

Methods	Spider 2.0-Snow	Spider 2.0-Lite
DIN-SQL [35]	0.0	4.8
DAIL-SQL [9]	6.6	8.8
Spider-Agent [16]	10.6	13.7
ReFoRCE [8]	38.0	29.6
RSL-SQL [3]	-	30.5
LinkAlign [50]	-	33.1
DeepEye-SQL (Ours)	**50.5**	**38.2**

that our software-grounded framework establishes a new benchmark for Text-to-SQL generation. Specifically, when integrated with the `Qwen3-Coder-30B-A3B` model, DeepEye-SQL achieves an execution accuracy of **73.5%** on BIRD-Dev. This result not only significantly outperforms all existing prompting-based baselines but, more remarkably, it also surpasses the leading fine-tuning-based systems like XiYan-SQL (73.3%) and CHASE-SQL (73.0%). It is crucial to note that these competing methods rely on substantially larger and often proprietary models (*e.g.,* GPT-4o and Gemini-1.5-Pro). On the official BIRD held-out Test set, DeepEye-SQL achieves **75.07%** execution accuracy with open-source models. Furthermore, scaling up to `Gemini-3-Pro` yields **76.58%** (Global Rank 5th) even with minimal sampling ($N = 1$). On the Spider-Test dataset, DeepEye-SQL with `Qwen3-Coder-30B-A3B` achieves an execution accuracy of **89.8%**. This performance matches the best fine-tuned method, OmniSQL, and surpasses other strong competitors like XiYan-SQL (89.7%). This demonstrates that our prompting-based framework can attain the same level of accuracy as highly specialized, fine-tuned models on this foundational benchmark without incurring any training costs. Moreover, on the newly released enterprise-grade Spider 2.0 benchmark (Table 4), DeepEye-SQL demonstrates superior robustness in handling complex, real-world constraints. On the rigorous "Snow" subset, DeepEye-SQL achieves **50.5%** execution accuracy, surpassing the specialized agentic framework ReFoRCE (38.0%) by a large margin (+12.5%). Notably, traditional prompting methods like DIN-SQL fail completely on this task (0.0%), highlighting that our structured engineering lifecycle is essential for navigating the ambiguity of realistic data warehousing scenarios. Furthermore, the consistently high performance of DeepEye-SQL across all three tested open-source models—71.1% with `Gemma3-27B`, 70.9% with `Qwen2.5-Coder-32B`, and 73.5% with `Qwen3-Coder-30B` on BIRD, which underscores the robustness and generalizability of our framework (Table 5).

RQ2: Is the performance gain of DeepEye-SQL attributable to its architectural design rather than the power of the underlying base model?

To isolate our framework's contribution from the base model's intrinsic capabilities, we evaluated it against leading prompting-based frameworks on identical open-source models (Table 5). The results confirm that DeepEye-SQL's architecture consistently provides the most substantial performance improvement. For instance, on `Gemma3-27B-Instruct`, DeepEye-SQL improves the CoT baseline by +16.9%, significantly outpacing the gains from CHESS (+11.9%) and Alpha-SQL (+14.4%). This trend holds across all tested models, culminating in a state-of-the-art 73.5% EX on `Qwen3-Coder-30B-A3B` (+11.8% gain).

Table 5. Performance comparison using the same LLM backbone. Gemma3-27B, Qwen2.5-32B, and Qwen3-30B denote Gemma3-27B-Instruction, Qwen2.5-Coder-32B-Instruct, and Qwen3-Coder-30B-A3B-Instruct, respectively.

Methods	Gemma3-27B		Qwen2.5-32B		Qwen3-30B	
	EX	Δ	EX	Δ	EX	Δ
CoT-Baseline	54.2	–	61.3	–	61.7	–
CHESS (IR, CG, UT)	66.1	+11.9	67.7	+6.4	67.9	+6.2
Alpha-SQL	68.6	+14.4	69.7	+8.4	71.2	+9.5
DeepEye-SQL (Ours)	**71.1**	**+16.9**	**70.9**	**+9.6**	**73.5**	**+11.8**

Table 6. Schema linking analysis on the BIRD-Dev dataset.

Schema Linking Methods	Table Recall (%)	Column Recall (%)	Avg. Tokens
No Schema Linking	–	–	5486.2
Direct Schema Linking	94.2	80.9	454.5
Reversed Schema Linking	97.0	94.0	495.9
Value Schema Linking	47.3	18.0	262.0
Robust Schema Linking	**98.1**	**95.4**	**627.4**

Question: Which accounts placed orders for *"household payment"* in Pisek?
Evidence: `k_symbol = 'SIPO'` refers to household payment.

LLM-based Linking (Direct & Reversed):

> *Found:* `account.account_id`, `district.district_id`, `order.k_symbol`, ...

X **Missed:** The essential column `trans.k_symbol`.

Value-based Linking:

✓ **Recovered:** `trans.k_symbol` by linking to value 'SIPO'.

```
-- Gold SQL
SELECT DISTINCT T2.account_id FROM trans AS T1 JOIN account AS T2 ON T1.account_id = T2.
account_id JOIN district AS T3 ON T2.district_id = T3.district_id WHERE T1.k_symbol = '
SIPO' AND T3.A2 = 'Pisek'
```

Fig. 4. A case study from BIRD-Dev (QID: 142) where Value-based Linking recovers a critical column (`trans.k_symbol`) missed by purely LLM-based linking methods.

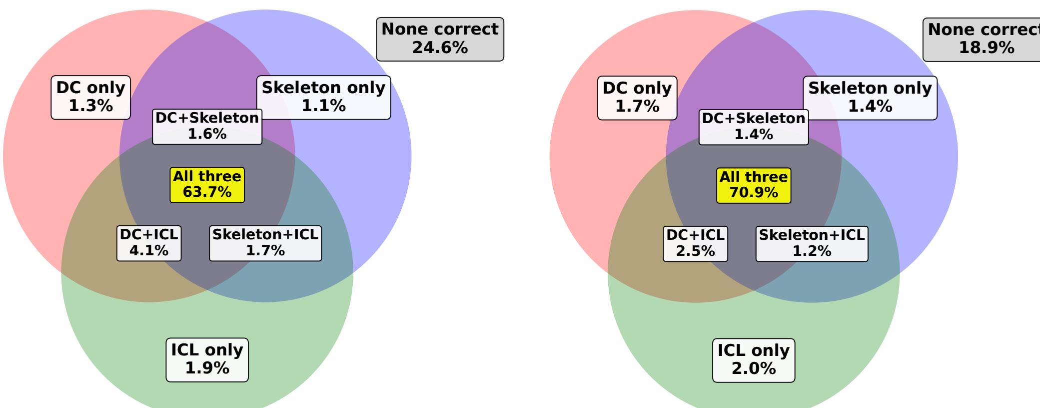
4.3 Robust Schema Linking Analysis

RQ3: How do the individual components of our Robust Schema Linking module contribute to its overall effectiveness and efficiency?

We analyzed our *Robust Schema Linking* module's components on BIRD-Dev (Table 6), measuring schema recall and token efficiency. The full approach achieves the highest recall (**98.1%** table, **95.4%** column), providing a critical foundation for SQL generation. It also dramatically boosts efficiency, reducing the input context by 9× from 5486.2 to 627.4 tokens compared to using the full schema.

Table 7. Analysis of SQL generation methods on the BIRD-Dev dataset.

SQL Generation Method	EX (%)	UB-EX (%)
Skeleton-based SQL Generation	69.2	77.9
ICL-based SQL Generation	70.9	78.3
D&C-based SQL Generation	70.3	78.5
N-version Programming for SQL Generation	**71.7**	**81.1**



(a) EX Correctness Overlap.

(b) UB-EX Correctness Overlap.

Fig. 5. Correctness overlap analysis of three SQL generation methods on BIRD-Dev dataset.

Analyzing the components reveals complementary strengths. Our novel *Reversed Schema Linking* (97.0% table, 94.0% column) significantly outperforms *Direct Schema Linking*, especially in column recall (+13.1%). This validates our hypothesis that prompting for a draft query is a more natural and effective reasoning method for LLMs than explicit extraction.

Finally, *Value-based Schema Linking*, while low in overall recall, is not a standalone linker. It acts as a precise mechanism to link columns via their data values, catching omissions from other methods. For instance (Figure 4), it correctly identified `trans.k_symbol` by linking the query's "household payment" to the data value `'SIPO'`. This bridged a semantic gap that schema-level reasoners (Direct and Reversed) missed, as they were confused by an ambiguous `k_symbol` name. This recovery proves the value-based method's essential role in our fault-tolerant design.

4.4 Analysis on N-Version Programming for SQL Generation

RQ4: How do the different SQL generators contribute to the overall performance, and does the N-version programming approach provide a tangible benefit?

We analyzed our three SQL generators' individual and combined performance (EX and UB-EX) on BIRD-Dev (Table 7, Figure 5). While all generators perform well individually (peaking at 70.9% EX), combining them in our N-version module already yields a 71.7% EX. The true strength of this approach, however, lies in its potential: the combined UB-EX reaches **81.1%**, a significant +2.6% gain over the best individual generator's potential (78.5%). This UB-EX increase confirms that the generators possess crucial diversity, solving different subsets of problems where one's failure is covered by another's success. The overlap analysis (Figure 5b) corroborates this: while 70.9% of

Table 8. Sensitivity analysis of example quality with Qwen3-Coder-30B-A3B on the BIRD-Dev dataset.

Generator	**Retrieval Strategy**	**SQL Jaccard**	**EX (%)**	**UB-EX (%)**
Single ICL-based	Least Relevant	0.44	70.1	75.8
	Most Relevant	0.58	70.9	78.3
Hybrid (ICL+D&C+Skl)	Least Relevant	0.44	71.5	80.2
	Most Relevant	0.58	71.7	81.1
DeepEye-SQL (Full Pipeline)	Least Relevant	0.44	73.4	80.8
	Most Relevant	0.58	**73.5**	**81.6**

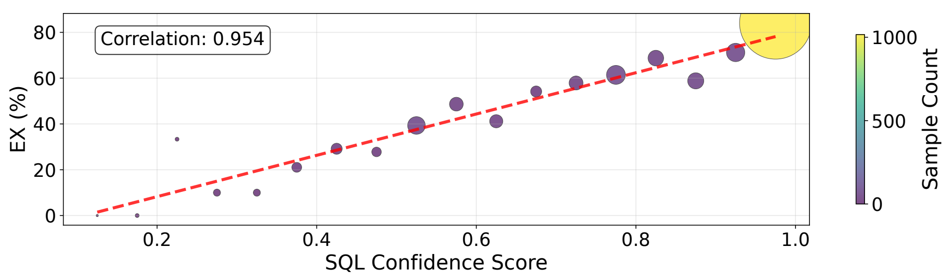


Fig. 6. Execution accuracy *vs.* SQL confidence on BIRD-Dev dataset with Qwen3-Coder-30B-A3B model.

correct answers are found by all three generators, a significant **10.2%** are solved by only one or two. This proves the generators are not redundant. This engineered diversity—finding a correct answer for 81.1% of questions—validates our fault-tolerant design and provides the rich candidate pool essential for achieving state-of-the-art performance

RQ5: Is the N-version programming robust to variations in few-shot example quality?

To answer this, we explicitly tested the system's sensitivity by manipulating the retrieval strategy. Instead of the standard approach that selects the top-K examples with the highest similarity ("Most Relevant"), we deliberately retrieved the K examples with the lowest similarity ("Least Relevant"). We quantified the quality of these examples using the Jaccard similarity between the example SQL and the ground truth (following DAIL-SQL [9]). As shown in Table 8, the "Least Relevant" set indeed exhibits significantly lower quality (0.44 vs. 0.58). Under this challenging condition, the performance of the standalone ICL generator drops noticeably (70.9% $\rightarrow$ 70.1%). However, our Hybrid Generator dampens this drop (71.7% vs. 71.5%), and the full DeepEye-SQL demonstrates remarkable stability (73.5% vs. 73.4%), verifying that our SE-inspired orchestration effectively masks the volatility of retrieval quality.

4.5 Confidence-aware SQL Selection Analysis

RQ6: Effectiveness of Confidence-aware Selection.

We first validate that voting reliability strongly correlates with consensus. As shown in Figure 6, we observe a high Pearson correlation of **0.954** between voting confidence and execution accuracy on BIRD-Dev. This confirms that while high-confidence votes are reliable (justifying our efficient "shortcut"), low-confidence scenarios require the deeper re-evaluation our method provides. Comparative results in Table 9 demonstrate that our approach consistently outperforms traditional

Table 9. Performance comparison of different SQL selection methods on the BIRD-Dev and Spider datasets.

Selection Method	BIRD-EX (%)	Spider-EX (%)
Gemma3-27B-Instruct		
Consistency-based voting	70.1	88.3
Confidence-aware selection	**71.1** (↑1.0)	**88.9** (↑0.6)
Qwen2.5-Coder-32B-Instruct		
Consistency-based voting	70.2	88.2
Confidence-aware selection	**70.9** (↑0.7)	**88.7** (↑0.5)
Qwen3-Coder-30B-A3B-Instruct		
Consistency-based voting	72.7	89.7
Confidence-aware selection	**73.5** (↑0.8)	**89.8** (↑0.1)

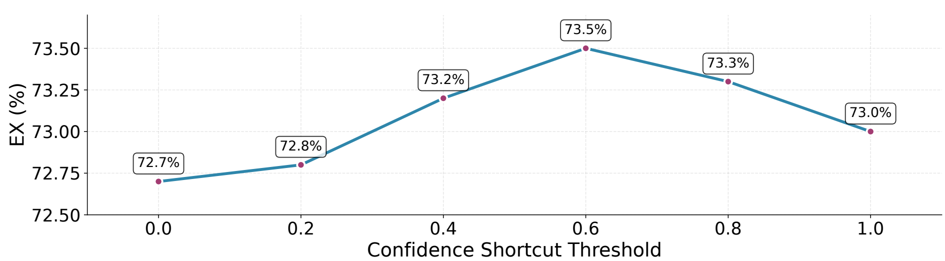


Fig. 7. Execution accuracy *vs.* Confidence Shorcut Threshold on BIRD-Dev dataset with Qwen3-Coder-30B-A3B model.

Consistency-based Voting [9, 18]. For instance, with `Gemma3-27B`, we achieve a 1.0% gain (70.1% → 71.1%) on BIRD, proving that selectively reviewing ambiguous cases yields superior robustness over blind majority voting.

RQ7: Sensitivity to Confidence Threshold (θ_{conf}).

We evaluated the robustness of the threshold θ_{conf}, which balances efficiency against rigorous review. Figure 7 shows that performance remains highly stable across a wide range: accuracy stays above 73.0% for $\theta_{conf} \in [0.4, 1.0]$, peaking at 73.5% when $\theta_{conf} = 0.6$. This indicates that our framework is not sensitive to specific hyperparameter choices.

4.6 Ablation Study

RQ8: What is the contribution of each key component to the overall performance of the DeepEye-SQL framework?

To understand the impact of each module, we conducted an ablation study on the BIRD-Dev dataset by progressively removing one component at a time. The results are detailed in Table 10.

The primary conclusion is that every component makes a positive and integral contribution, as removing any single module degrades performance. The *ICL-based SQL Generation* is the most critical individual module, with its removal causing the largest performance drop of **2.5%**. *Semantic Value Retrieval* and *SQL Unit Testing and Revision via Tool-Chain* are also highly impactful, each accounting for a **2.1%** gain. This highlights the necessity of grounding the LLM in real data and

Table 10. Ablation study of DeepEye-SQL with Qwen3-Coder-30B-A3B on the BIRD-Dev dataset.

Variant	EX (%)	Δ (%)
DeepEye-SQL	**73.5**	–
w/o Semantic Value Retrieval	71.4	-2.1
w/o Robust Schema Linking	71.8	-1.7
w/o Skeleton-based Generation	72.2	-1.3
w/o ICL-based Generation	71.0	-2.5
w/o D&C-based Generation	72.3	-1.2
w/o Tool-Chain Testing & Revision	71.4	-2.1
w/o Confidence-aware Selection	72.7	-0.8

Table 11. Efficiency comparison on the BIRD-Dev dataset. DeepEye-SQL achieves orders-of-magnitude lower latency than search-based (Alpha-SQL) and linear-chain (CHESS) baselines while maintaining higher accuracy.

Method	Avg. Input Tokens (K)	Avg. Output Tokens (K)	Avg. Latency (s/query)	EX (%)
CHESS (IR, SS, CS)	327.02	27.83	284.4	67.9
Alpha-SQL	138.03	72.21	377.1	71.2
DeepEye-SQL	**23.21**	**23.16**	**17.8**	**73.5**
Semantic Value Retrieval	0.67	0.03	–	–
Robust Schema Linking	13.91	6.33	–	–
N-version Generation	5.28	11.38	–	–
Tool-Chain Verification	3.16	5.41	–	–
Confidence-aware Selection	0.19	0.01	–	–

externalizing the debugging process. The significant contributions from the SQL generation modules and the selection mechanism confirm the value of our N-version programming and confidence-aware selection strategies. Overall, the results confirm that the high performance of DeepEye-SQL is not due to any single component, but rather the synergistic collaboration of all modules in its carefully designed pipeline.

4.7 Efficiency Analysis

RQ9: How does DeepEye-SQL perform in terms of efficiency and scalability compared to SOTA baselines?

To answer this, we conducted a comprehensive evaluation covering token cost, end-to-end latency, and system scalability.

(i) Cost Efficiency (Token Consumption). As shown in Table 11, DeepEye-SQL demonstrates superior token efficiency. Compared to Alpha-SQL [18] and CHESS [46], our framework reduces input token consumption by nearly **6×** (138.03K vs. 23.21K) and **14×** (327.02K vs. 23.21K), respectively, while achieving higher accuracy.

(ii) Time Efficiency (Latency). Beyond cost, time-to-result is critical for real-world deployment. Table 11 reveals that DeepEye-SQL operates with an average latency of **17.8s** per query, which is orders of magnitude faster than search-centric baselines like Alpha-SQL (377.1s, ~21× slower) and linear chains like CHESS (284.4s, ~16× slower). The high latency of Alpha-SQL stems from its

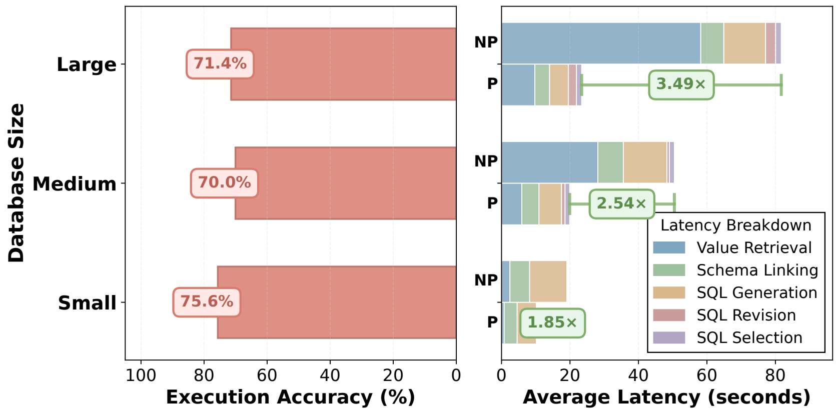


Fig. 8. System Scalability Analysis on BIRD-Dev. (Left) Performance Robustness: We categorize databases into Small, Medium, and Large based on file size percentiles. DeepEye-SQL maintains consistent execution accuracy even on Large schemas. (Right) Parallel Execution Efficiency: Comparison between Parallel (P) and Non-Parallel (NP) execution modes. The parallel pipeline significantly reduces latency, achieving a **3.49×** speedup on Large databases by overlapping I/O-bound stages with Generation.

sequential Monte Carlo Tree Search (MCTS), which inherently blocks parallelization. In contrast, DeepEye-SQL's *N-Version Programming* and *Tool-Chain* are designed for parallel and conditional execution, maximizing throughput.

(iii) Scalability Analysis. To further assess robustness, Figure 8 breaks down performance by database complexity.

- **Robust Accuracy (Figure 8 Left):** DeepEye-SQL maintains high execution accuracy across Small, Medium, and Large schemas, showing no significant degradation even as complexity increases.
- **Parallel Acceleration (Figure 8 Right):** The benefit of our parallel architecture scales with data size. On Large databases, enabling parallel execution yields a **3.49× speedup** (reducing latency from ~80s to ~23s) by effectively overlapping I/O-bound stages with Generation.

4.8 Error Analysis

RQ10: What are the dominant failure modes of DeepEye-SQL, and how does its error distribution compare to SOTA baselines?

To gain deeper insights into the limitations of DeepEye-SQL, we performed a manual inspection of failure cases on the BIRD-Dev set. We compared DeepEye-SQL with OmniSQL [19] and Alpha-SQL [18] using a fine-grained taxonomy covering five error dimensions, each divided into specific subclasses: **Schema** (① Selection, ② Join), **Value** (① Mapping, ② Format), **Function** (① Aggregation, ② Calculation), **Logic** (① Filter, ② Presentation), and **Ground-truth** (① Gold Error, ② Ambiguity).

Figure 9 illustrates the comparative distribution of these errors. DeepEye-SQL demonstrates a substantial advantage in handling *Value-related* errors (only 17 instances), significantly outperforming baselines (56 and 39). This attributes to our *Semantic Value Retrieval* strategy which effectively resolves both ① *Value Mapping* and ② *Data Format* issues. Furthermore, the lower error rate in **Logic** and **Function** categories (67 and 75 respectively) validates the synergy of our *N-version SQL Generation* and *SQL Unit Testing & Revision* modules. However, **Schema Linking** remains the most challenging category for all models (126 errors for DeepEye-SQL). Failures here are dominated by

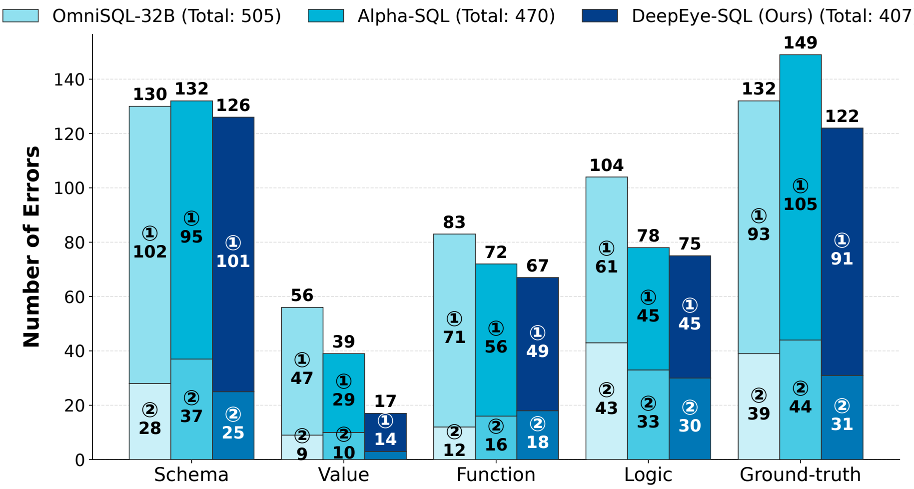


Fig. 9. Comparative Error Analysis on BIRD-Dev.

incorrect ① *Column Selection* and ② *Join Paths*, typically triggered by complex multi-hop requirements or obscure column names. We also observed that roughly 30% of the reported failures (122 cases) fall into the **Ground-truth** category, stemming from ambiguity in questions or errors in the gold SQLs, indicating the ceiling imposed by dataset quality.

5 Related Work

Text-to-SQL Solutions. The task of translating natural language into executable SQL queries has evolved significantly [22]. The recent emergence of Large Language Models (LLMs) has marked a new paradigm. Current research on applying LLMs to Text-to-SQL is largely divided into two categories: fine-tuning and prompting-based methods. Fine-tuning methods adapt open-source models like Code Llama or Qwen for the Text-to-SQL task by training them on large corpora of question-SQL pairs [19, 20, 24, 43, 56]. This approach can yield highly efficient and specialized models but may exhibit limited generalization to out-of-domain or highly complex scenarios. In contrast, prompting-based methods leverage the powerful in-context learning and reasoning capabilities of very large, often proprietary, models like GPT-4o [12] and Gemini [1] without requiring model training. To manage the complexity of the task, these methods typically decompose the process into a multi-stage pipeline, including sub-tasks like schema linking, SQL generation, and refinement [17, 18, 30, 34, 35, 64]. While powerful, these frameworks often rely on a single generation path and the fallible self-correction abilities of the LLM, which can compromise robustness. Our work, DeepEye-SQL, belongs to the prompting-based category but differentiates itself by explicitly adopting a systematic framework inspired by software engineering.

Software Engineering. The challenge of building robust and reliable LLM-based systems often mirrors the complexities of traditional software development. In response, software engineering has established a set of core principles to manage complexity and ensure product quality. The *Software Development Life Cycle (SDLC)* [40] provides a foundational, systematic process for software creation, typically involving phases such as requirements analysis, implementation, testing, and deployment. To deconstruct existing systems and inform new designs, practitioners often employ *Reverse Engineering* [38], a process of analyzing a finished product to deduce its underlying specifications.

To enhance system reliability, fault tolerance techniques are critical. A notable example is *N-Version Programming* [4], where multiple, independently developed versions of a component are executed in parallel, and their results are adjudicated to mask faults and increase the likelihood of a correct outcome. For quality assurance, *Unit Testing* [39] is a fundamental practice. It involves the granular testing of individual software components or "units" in isolation to verify that each part functions correctly according to its design specifications. This bottom-up approach is crucial for identifying and rectifying defects early in the development process. Finally, to govern the release process, a *Quality Gate* [41] acts as a final checkpoint, enforcing a set of predefined criteria to determine whether a software artifact meets the required quality standard for deployment. Although central to traditional software engineering, these principles have seen limited systematic adoption in LLM-based pipelines. We bring them to Text-to-SQL to enable a structured, verifiable, and robust pipeline.

6 Conclusion

In this paper, we present DeepEye-SQL, a verifiable SDLC-style workflow for Text-to-SQL that targets system-level reliability through end-to-end correctness. Without any fine-tuning, DeepEye-SQL leverages open-source MoE LLMs (about 30B total parameters with roughly 3B activated) and achieves 73.5% execution accuracy on BIRD-Dev, 75.07% on the official BIRD-Test leaderboard, and 89.8% on Spider-Test, outperforming state-of-the-art approaches that rely on larger models or extensive training. Overall, these results suggest that principled orchestration, rather than fine-tuning or scaling up inference alone, is a promising path toward reliable Text-to-SQL.

Acknowledgments

This paper was supported by the NSF of China (62402409); Youth S&T Talent Support Programme of Guangdong Provincial Association for Science and Technology (SKXRC2025461); the Young Talent Support Project of Guangzhou Association for Science and Technology (QT-2025-001); Guangdong Basic and Applied Basic Research Foundation (2023A1515110545); Guangzhou Basic and Applied Basic Research Foundation (2025A04J3935); and Guangzhou-HKUST(GZ) Joint Funding Program (2025A03J3714).

References

[1] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. *CoRR* abs/2312.11805 (2023). arXiv:2312.11805 doi:10.48550/ARXIV.2312.11805

[2] Jinheon Baek, Horst Samulowitz, Oktie Hassanzadeh, Dharmashankar Subramanian, Sola Shirai, Alfio Gliozzo, and Debarun Bhattacharjya. 2025. Knowledge Base Construction for Knowledge-Augmented Text-to-SQL. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 26569–26583. https://aclanthology.org/2025.findings-acl.1363/

[3] Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. RSL-SQL: Robust Schema Linking in Text-to-SQL Generation. *CoRR* abs/2411.00073 (2024). arXiv:2411.00073 doi:10.48550/ARXIV.2411.00073

[4] Liming Chen and Algirdas Avizienis. 1978. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, Vol. 1. 3–9.

[5] chroma core. 2025. *The AI-native open-source embedding database.* https://github.com/chroma-core/chroma Accessed: 2025-10-17.

[6] Yeounoh Chung, Gaurav Tarlok Kakkar, Yu Gan, Brenton Milne, and Fatma Ozcan. 2025. Is Long Context All You Need? Leveraging LLM's Extended Context for NL2SQL. *Proc. VLDB Endow.* 18, 8 (2025), 2735–2747.

[7] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR* abs/2501.12948 (2025). arXiv:2501.12948 doi:10.48550/ARXIV.2501.12948

[8] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. ReFoRCE: A Text-to-SQL Agent with Self-Refinement, Format Restriction, and Column Exploration. *CoRR* abs/2502.00675 (2025). arXiv:2502.00675 doi:10.48550/ARXIV.2502.00675

[9] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145. doi:10.14778/3641204.3641221

[10] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul J. Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. *ACM Comput. Surv.* 41, 2 (2009), 9:1–9:76. doi:10.1145/1459352.1459354

[11] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024). arXiv:2409.12186 doi:10.48550/ARXIV.2409.12186

[12] Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Madry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll L. Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, and Dane Sherburn. 2024. GPT-4o System Card. *CoRR* abs/2410.21276 (2024). arXiv:2410.21276 doi:10.48550/ARXIV.2410.21276

[13] Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean-Bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Róbert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucinska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju-yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle K. Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley,

Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clément Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry (Dima) Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. *CoRR* abs/2503.19786 (2025). arXiv:2503.19786 doi:10.48550/ARXIV.2503.19786

[14] Janet L. Kolodner. 1993. *Case-Based Reasoning*. Morgan Kaufmann. doi:10.1016/C2009-0-27670-7

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. doi:10.1145/3600006.3613165

[16] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=XmProj9cPs

[17] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? [Experiment, Analysis & Benchmark]. *Proc. VLDB Endow.* 17, 11 (2024), 3318–3331. doi:10.14778/3681954.3682003

[18] 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. In *Forty-second International Conference on Machine Learning*. OpenReview.net. https://openreview.net/forum?id=kGg1ndttmI

[19] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale. *Proc. VLDB Endow.* 18, 11 (2025), 4695–4709. https://www.vldb.org/pvldb/vol18/p4695-li.pdf

[20] 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. *Proc. ACM Manag. Data* 2, 3 (2024), 127. doi:10.1145/3654930

[21] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. 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 (2024).

[22] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going? *IEEE Trans. Knowl. Data Eng.* 37, 10 (2025), 5735–5754.

[23] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025. NL2SQL-BUGs: A Benchmark for Detecting Semantic Errors in NL2SQL Translation. *CoRR* abs/2503.11984 (2025). arXiv:2503.11984 doi:10.48550/ARXIV.2503.11984

[24] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025. XiYan-SQL: A Novel Multi-Generator Framework For Text-to-SQL. *CoRR* abs/2507.04701 (2025). arXiv:2507.04701 doi:10.48550/ARXIV.2507.04701

[25] Tianqi Luo, Chuhan Huang, Leixian Shen, Boyan Li, Shuyu Shen, Wei Zeng, Nan Tang, and Yuyu Luo. 2025. nvBench 2.0: Resolving Ambiguity in Text-to-Visualization through Stepwise Reasoning. *arXiv preprint arXiv:2503.12880* (2025).

[26] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Natural Language to SQL: State of the Art and Open Problems. *Proc. VLDB Endow.* 18, 12 (2025), 5466–5471.

[27] Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. DeepEye: Towards Automatic Data Visualization. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 101–112. doi:10.1109/ICDE.2018.00019

[28] Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021. Synthesizing Natural Language to Visualization (NL2VIS) Benchmarks from NL2SQL Benchmarks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1235–1247. doi:10.1145/3448016.3457261

[29] Yuyu Luo, Nan Tang, Guoliang Li, Jiawei Tang, Chengliang Chai, and Xuedi Qin. 2022. Natural Language to Visualization by Neural Machine Translation. *IEEE Trans. Vis. Comput. Graph.* 28, 1 (2022), 217–226. doi:10.1109/TVCG.2021.3114848

[30] Peixian Ma, Boyan Li, Runzhi Jiang, Ju Fan, Nan Tang, and Yuyu Luo. 2024. A Plug-and-Play Natural Language Rewriter for Natural Language to SQL. *CoRR* abs/2412.17068 (2024). arXiv:2412.17068 doi:10.48550/ARXIV.2412.17068

[31] 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. *CoRR* abs/2408.07702 (2024). arXiv:2408.07702 doi:10.48550/ARXIV.2408.07702

[32] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[33] Joel Ossher, Sushil Krishna Bajracharya, and Cristina Videira Lopes. 2010. Automated dependency resolution for open source software. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, Jim Whitehead and Thomas Zimmermann (Eds.). IEEE Computer Society, 130–140. doi:10.1109/MSR.2010.5463346

[34] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=CvGqMD5OtX

[35] 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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/72223cc66f63ca1aa59edaec1b3670e6-Abstract-Conference.html

[36] 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. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 11268–11292. https://aclanthology.org/2025.acl-long.552/

[37] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

[38] M. G. Rekoff. 1985. On reverse engineering. *IEEE Trans. Syst. Man Cybern.* 15, 2 (1985), 244–252. doi:10.1109/TSMC.1985.6313354

[39] Per Runeson. 2006. A Survey of Unit Testing Practices. *IEEE Softw.* 23, 4 (2006), 22–29. doi:10.1109/MS.2006.91

[40] Nayan B. Ruparelia. 2010. Software development lifecycle models. *ACM SIGSOFT Softw. Eng. Notes* 35, 3 (2010), 8–13.

[41] Joscha Schnell and Gunther Reinhart. 2016. Quality management for battery production: a quality gate concept. *Procedia CIRP* 57 (2016), 568–573.

[42] Lei Sheng and Shuai-Shuai Xu. 2025. CSC-SQL: Corrective Self-Consistency in Text-to-SQL via Reinforcement Learning. *CoRR* abs/2505.13271 (2025). arXiv:2505.13271 doi:10.48550/ARXIV.2505.13271

[43] Lei Sheng, Shuai-Shuai Xu, and Wei Xie. 2025. BASE-SQL: A powerful open source Text-To-SQL baseline approach. *CoRR* abs/2502.10739 (2025). arXiv:2502.10739 doi:10.48550/ARXIV.2502.10739

[44] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. 2025. Automatic Metadata Extraction for Text-to-SQL. *CoRR* abs/2505.19988 (2025). arXiv:2505.19988 doi:10.48550/ARXIV.2505.19988

[45] Zhihao Shuai, Boyan Li, Siyu Yan, Yuyu Luo, and Weikai Yang. 2025. DeepVIS: Bridging Natural Language and Data Visualization Through Step-wise Reasoning. *CoRR* abs/2508.01700 (2025). arXiv:2508.01700 doi:10.48550/ARXIV.2508.01700

[46] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. *CoRR* abs/2405.16755 (2024). arXiv:2405.16755 doi:10.48550/ARXIV.2405.16755

[47] Martyn Thomas and Frank E. McGarry. 1994. Top-Down vs. Bottom-Up Process Improvement. *IEEE Softw.* 11, 4 (1994), 12–13. doi:10.1109/52.300121

[48] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schroder-Preikschat, and Reiner Schmid. 2012. Eliminating single points of failure in software-based redundancy. In *2012 Ninth European Dependable Computing Conference*. IEEE, 49–60.

[49] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=1PL1NIMMrw

[50] Yihan Wang and Peiyu Liu. 2025. LinkAlign: Scalable Schema Linking for Real-World Large-Scale Multi-Database Text-to-SQL. *CoRR* abs/2503.18596 (2025). arXiv:2503.18596 doi:10.48550/ARXIV.2503.18596

[51] Yifan Wu, Jingze Shi, Bingheng Wu, Jiayi Zhang, Xiaotian Lin, Nan Tang, and Yuyu Luo. 2025. Concise Reasoning, Big Gains: Pruning Long Reasoning Trace with Difficulty-Aware Prompting. *CoRR* abs/2505.19716 (2025).

[52] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. *Proc. ACM Manag. Data* 3, 3 (2025), 194:1–194:24. doi:10.1145/3725331

[53] Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. HAIChart: Human and AI Paired Visualization System. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3178–3191.

[54] Yupeng Xie, Zhiyang Zhang, Yifan Wu, Sirong Lu, Jiayi Zhang, Zhaoyang Yu, Jinlin Wang, Sirui Hong, Bang Liu, Chenglin Wu, et al. 2025. Visjudge-bench: Aesthetics and quality assessment of visualizations. *arXiv preprint arXiv:2510.22373* (2025).

[55] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jian Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *CoRR* abs/2505.09388 (2025). arXiv:2505.09388 doi:10.48550/ARXIV.2505.09388

[56] Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing Text-to-SQL Data from Weak and Strong LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 7864–7875. doi:10.18653/V1/2024.ACL-LONG.425

[57] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. Association for Computational Linguistics, 3911–3921.

[58] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

[59] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating Agentic Workflow Generation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=z5uVAKwmjf

[60] Qingjie Zhang, Di Wang, Haoting Qian, Yiming Li, Tianwei Zhang, Minlie Huang, Ke Xu, Hewu Li, Liu Yan, and Han Qiu. 2025. Understanding the Dark Side of LLMs' Intrinsic Self-Correction. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 27066–27101. https://aclanthology.org/2025.acl-long.1314/

[61] Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025. Reward-SQL: Boosting Text-to-SQL via Stepwise Reasoning and Process-Supervised Rewards. *CoRR* abs/2505.04671 (2025).

[62] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *CoRR* abs/2506.05176 (2025). arXiv:2506.05176 doi:10.48550/ARXIV.2506.05176

[63] Yizhang Zhu, Shiyin Du, Boyan Li, Yuyu Luo, and Nan Tang. 2024. Are Large Language Models Good Statisticians?. In *NeurIPS*.

[64] Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. EllieSQL: Cost-Efficient Text-to-SQL with Complexity-Aware Routing. *CoRR* abs/2503.22402 (2025).

[65] Yizhang Zhu, Liangwei Wang, Chenyu Yang, Xiaotian Lin, Boyan Li, Wei Zhou, Xinyu Liu, Zhangyang Peng, Tianqi Luo, Yu Li, Chengliang Chai, Chong Chen, Shimin Di, Ju Fan, Ji Sun, Nan Tang, Fugee Tsung, Jiannan Wang, Chenglin Wu, Yanwei Xu, Shaolei Zhang, Yong Zhang, Xuanhe Zhou, Guoliang Li, and Yuyu Luo. 2025. A Survey of Data Agents: Emerging Paradigm or Overstated Hype? *CoRR* abs/2510.23587 (2025). arXiv:2510.23587 doi:10.48550/ARXIV.2510.23587

Received October 2025; revised January 2026; accepted February 2026