

KODE: Mining Implicit Domain Knowledge from Historical Experience for Text-to-SQL

Qingshuo Lyu^{1,2}, Jiangyi Yin^{1,2,*}, Zhao Li^{1,2},
Meijie Du^{1,2}, Zhuojun Jiang^{1,2}, Qingyun Liu^{1,2}

¹Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

²School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract

Text-to-SQL systems built on large language models still falter in domain deployments, where users underspecify queries while databases enforce rigid schemas. This gap often reflects implicit knowledge—hidden joins, derived formulas, and domain-specific filters—that natural language (NL) omits but SQL must encode. We argue that validated historical (NL, SQL) pairs make this knowledge retrievable: NL states the semantic goal while SQL formalizes the execution logic. Treating them as dual views of one program, we propose KODE-SQL, a multi-agent framework for deployments where in-domain traces are available or gradually accumulate. KODE-Miner aligns NL and SQL graphs and distills unmatched SQL regions into reusable templates; KODE-Solver applies them through rule-then-retrieval generation; and KODE-Critic verifies candidates and confidence-filters new traces for later queries. KODE-SQL improves execution accuracy by up to 11.1 points on ScienceBenchmark, yields consistent plug-in gains (+4–7 points), and delivers consistent improvements on schema-disjoint benchmarks through abstract-template transfer.

1 Introduction

Text-to-SQL systems promise to democratize data access by letting non-experts query relational databases in natural language, and have become a key building block of modern business intelligence and enterprise data services (Hong et al., 2025; Wang et al., 2026; Yang et al., 2025). Driven by large language models (LLMs), recent systems have substantially advanced execution accuracy on standard benchmarks (Zhang et al., 2025a; Zhu et al., 2024). However, the gap between user intent and executable database logic remains far from closed.

*Corresponding author.

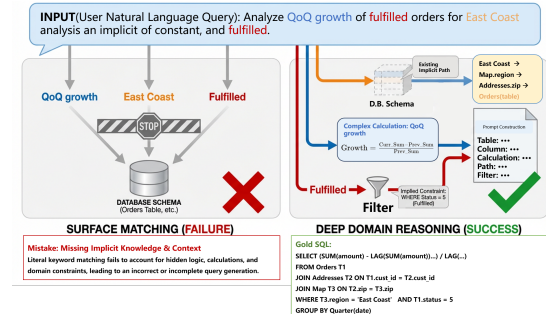


Figure 1: An illustration of the semantic gap grounded in a real-world analytics scenario. While surface matching fails on ambiguous queries due to missing direct links (Left), deep reasoning (Right) successfully bridges gaps by resolving three latent structures: *implicit topology* (silent tables), *complex calculations* (derived metrics), and *default filters* (hidden domain constraints).

The persistent obstacle is a *semantic gap* between underspecified natural language (NL) and the rigid, fully-specified structure of SQL (Chen et al., 2025; Hong et al., 2024; Fürst et al., 2024). As illustrated in Figure 1, a query about “QoQ growth” exposes three recurring failure modes: **missing topology** (intermediate tables required to connect entities are silently dropped), **missing computation** (derived metrics such as “QoQ growth” are not unfolded into the correct arithmetic), and **missing constraints** (e.g., “fulfilled” requires status = 5). These symptoms share a common root: SQL encodes *implicit knowledge*—join paths, computational formulas, and domain constraints—that the NL surface form never states, and the solver has no way to recover it from the query alone.

Where should this knowledge come from? Historical (NL, SQL) pairs are the natural source—each pair is a validated record in which the SQL side already spells out what the NL side omits. Yet existing methods see this data but rarely isolate the knowledge inside it: prompting and few-shot approaches (Li et al., 2025; Xie et al., 2024) pri-

marily retrieve whole demonstrations; experience and semantic-memory methods (Chu et al., 2024; Biswal et al., 2026) retain reusable interaction-level experience; synthetic-data and rule-based methods (Zhou et al., 2022; Pourreza et al., 2025) bypass real pairs and incur hallucination or annotation cost; and graph-augmented methods (Peng et al., 2026; Sheng and Shuai, 2025) build graphs only on the NL or schema side, leaving SQL itself as an opaque target. KODE-SQL instead treats a trace as evidence from which to distill the SQL substructures not grounded in NL, rather than treating the full trace as the atomic memory unit.

Our perspective. Each historical (NL, SQL) pair describes the same query in two languages: NL states the goal, SQL states the steps. The gap between them—unstated joins, hidden formulas, default filters—is exactly the implicit knowledge that solvers miss. We therefore view NL and SQL as two views of one program, and recover the missing knowledge by aligning them in a shared graph: matched parts give surface mappings, unmatched parts expose domain rules.

Our approach. We propose **KODE-SQL**, a graph-grounded framework for log-rich or gradually accumulating in-domain deployments. **KODE-Miner** aligns historical NL and ground-truth SQL graphs, then distills unaligned SQL regions into domain rules and structural exemplars. **KODE-Solver** prioritizes deterministic rule matches over dense retrieval and selects candidates by execution-aware consensus. **KODE-Critic** detects graph conflicts, rewrites only affected substructures, and optionally promotes high-confidence outputs for future queries, enabling confidence-filtered deployment-time knowledge accumulation.

Contributions. Our contributions are threefold:

(1) **A new perspective.** We observe that implicit domain knowledge—hidden joins, derived formulas, default filters—is not a missing resource to be supplied externally, but a structural signal already present in every historical (NL, SQL) pair: what NL omits is exactly what SQL must encode. This reframes the source of domain knowledge from external supervision to intrinsic NL–SQL mismatch.

(2) **A unified extract–use–verify framework.** KODE-SQL mines knowledge from historical pairs, applies it through stratified retrieval, and verifies outputs over the same graph representation. Confidence-filtered backfill allows validated deployment traces to benefit later queries while controlling error propagation.

Table 1: Module-level input–output flow of KODE-SQL.

Module	Input	Output
Miner	validated NL–SQL logs	rules + exemplars
Solver	query, schema, knowledge	candidate SQL
Critic	candidate, graph evidence	revised SQL + backfill

(3) **Evidence across three deployment regimes.** KODE-SQL gains up to **11.1 EX points** on knowledge-heavy ScienceBenchmark and boosts six off-the-shelf solvers. Controlled tests separate implicit knowledge from SQL imitation, show modest transfer of abstract templates to unseen schemas, and demonstrate accuracy growth under leakage-free deployment-time accumulation.

2 Method

Scope and knowledge distinction. KODE-SQL targets deployments with validated in-domain traces or traces that gradually accumulate. With an empty knowledge base, it falls back to its underlying retrieval-augmented solver. Given an aligned historical pair, *explicit knowledge* is SQL content directly grounded by aligned NL evidence (a mentioned entity, value, or operation); *implicit knowledge* is a connected SQL subgraph containing at least one node without NL grounding, such as an omitted bridge join, derived formula, default constraint, or set-operation structure. For example, if “fulfilled orders” aligns to an SQL filter status=5, the mentioned order entity is explicit while the unstated status mapping is mined as implicit domain knowledge.

2.1 KODE-Miner: Knowledge Alignment Agent

KODE-Miner bridges the gap between an underspecified query q and its SQL s in three stages: graph projection (§2.1.1), implicit region localization (§2.1.2), and template extraction (§2.1.3); the results are organized into two knowledge resources (§2.1.4).

2.1.1 Graph Representation

Natural language queries are often underspecified, while SQL requires a complete execution plan. To align the two views, we represent both the user query q and the SQL program s as graphs and compute an optimal partial alignment in a shared semantic space.

(1) User Query Understanding and Graph Representation

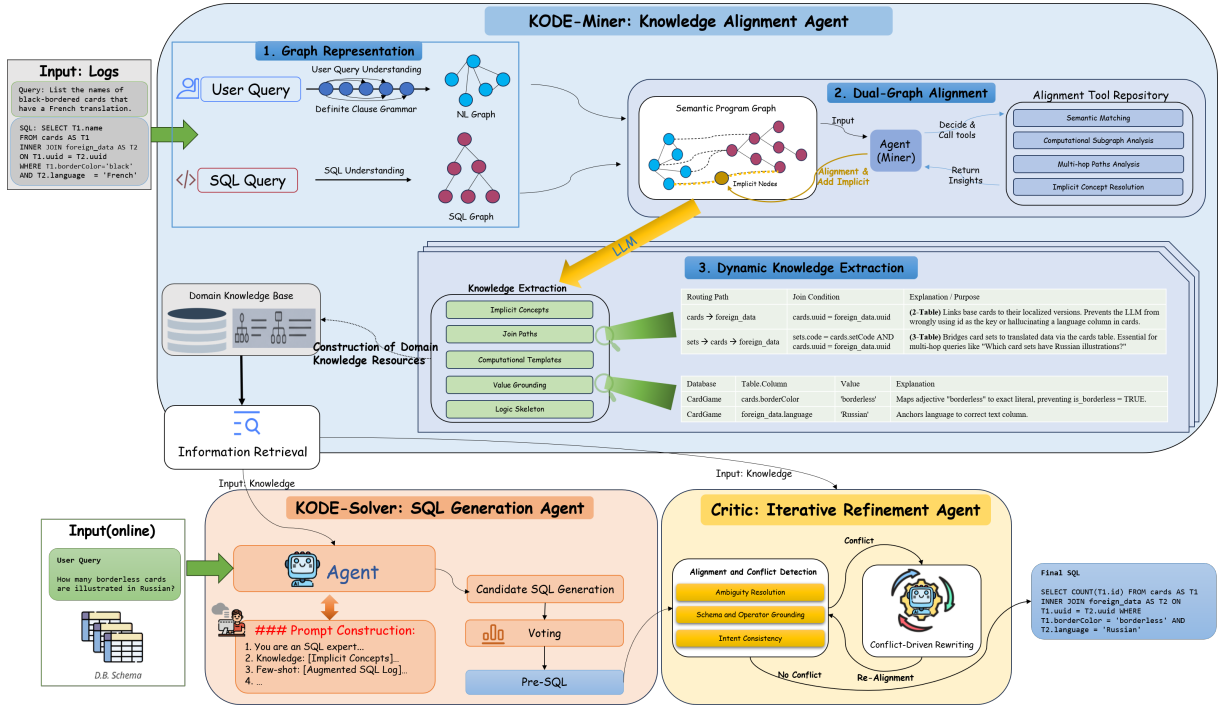


Figure 2: Overview of KODE-SQL. A framework where (1) KODE-Miner extracts domain knowledge via NL-SQL graph alignment; (2) KODE-Solver applies these priors for retrieval-augmented SQL generation; and (3) KODE-Critic verifies the generated candidates.

We map the query q into a graph G_q that preserves *evidence* rather than executable logic:

$$G_q = (V_q, E_q), \quad V_q = V_q^{\text{ent}} \cup V_q^{\text{op}}. \quad (1)$$

Nodes. V_q^{ent} covers three entity categories—*query objects* (e.g., “movie”), *constants* (temporal/numeric/categorical literals, e.g., “after 2010,” “top 10”), and *qualifiers* (e.g., “action”); V_q^{op} captures operational intent (e.g., “filter,” “exclude”).

Edges. E_q contains directed edges of two types: (i) *attribution edges* ($\rightarrow_{\text{attr}}$) link query objects to express belonging or composition (e.g., *director* $\rightarrow_{\text{attr}}$ *movie*); and (ii) *modification edges* ($\rightarrow_{\text{mod}}$) link a constant or qualifier to the object it restricts (e.g., *after 2020* $\rightarrow_{\text{mod}}$ *movie*).

G_q retains uncertain modifiers by design, deferring their resolution to alignment (prompts in Appendix D.2).

(2) SQL Understanding and Graph Representation

To go beyond surface token matching, we decompose s into a normalized semantic graph G_s built upon the abstract syntax tree (Aho et al., 2006) and relational operator tree (Begoli et al., 2018) paradigms. We further define a finite set of atomic operators with typed slots and structural constraints,

so that G_s represents operators and schema elements as nodes and data-flow and slot-filling dependencies as edges. The complete operator and slot specifications are given in the Appendix D.1.

This representation makes SQL semantics explicit by decomposing the query into operator-level units, allowing operators and slots in G_s to align with evidence nodes in G_q .

2.1.2 Dual-View Graph Alignment

Surface mappings are already well handled by LLMs; the difficulty lies in *implicit knowledge*—domain rules, unstated joins, derived metrics—that q omits but s must encode. We first anchor easy matches, then expose the implicit regions against these anchors.

(1) Explicit Anchor Alignment. We prompt an LLM to propose candidate pairs $(v, u) \in V_q \times V_s^{\text{fill}}$. Pairs that satisfy slot-type compatibility are kept, forming a partial alignment A^* . A^* is a scaffold rather than the final target: it grounds the obvious matches so that the unmatched regions become the focus of the next stage.

(2) Semantic Program Graph. We lift A^* into a unified graph that retains every SQL node together with its alignment state, called the *Semantic*

Program Graph G_{align} :

$$\begin{aligned} G_{\text{align}} &= (V_{\text{align}}, E_{\text{align}}), \\ V_{\text{align}} &= \{(u, \mathcal{M}(u)) \mid u \in V_s\}, \\ E_{\text{align}} &= \{(u_i, u_j) \in E_s \mid \\ &\quad \text{Con}(u_i, u_j) \wedge \text{Chk}(u_i, u_j)\}. \end{aligned} \quad (2)$$

where $\mathcal{M}(u)$ records whether u is grounded by NL evidence: $\mathcal{M}(u) = v$ if some $(v, u) \in A^*$, and $\mathcal{M}(u) = \varepsilon$ otherwise. Edges from E_s are kept only when they satisfy path reachability (Con) and type consistency (Chk).

Based on $\mathcal{M}(\cdot)$, SQL nodes split into **explicit nodes** $V_s^{\text{ex}} = \{u \mid \mathcal{M}(u) \neq \varepsilon\}$, which serve as structural anchors, and **implicit nodes** $V_s^{\text{im}} = \{u \mid \mathcal{M}(u) = \varepsilon\}$, which mark where the query is underspecified.

(3) Implicit Region Localization. Since a lone implicit node is rarely meaningful, we define implicit knowledge at the subgraph level:

Definition (Implicit Knowledge K_{im}). The set of connected subgraphs of G_{align} that contain at least one implicit node:

$$K_{\text{im}} = \{G_{\text{sub}} \subseteq G_{\text{align}} \mid V(G_{\text{sub}}) \cap V_s^{\text{im}} \neq \emptyset\}. \quad (3)$$

Each G_{sub} isolates a coherent piece of hidden logic, falling into two cases: (i) the logic has an NL counterpart missed by direct matching (e.g., “net profit” $\leftrightarrow$ revenue - cost), recovered by pairing G_{sub} with the residual NL evidence $R_q = V_q \setminus \{v \mid \exists u, (v, u) \in A^*\}$ via an LLM semantic check; (ii) the logic is SQL-only (e.g., implicit joins, type casts, business constraints), retained as-is for downstream interpretation.

Finally, we prompt an LLM with each G_{sub} , its schema context, and the matched residual NL evidence to produce a short structured rationale (e.g., “*filter by status='active' because the query mentions 'active users'*”). The rationale is attached to G_{sub} as a property node, turning the Semantic Program Graph into an interpretable representation ready for knowledge extraction.

2.1.3 Dynamic Knowledge Extraction

From each $G_{\text{align}}^{(i)}$ we extract five complementary templates that disentangle distinct sources of NL–SQL mismatch:

$$\mathcal{E}(G_{\text{align}}^{(i)}) = \{T_C^{(i)}, T_V^{(i)}, T_P^{(i)}, T_F^{(i)}, T_L^{(i)}\}. \quad (4)$$

(1) Implicit Concepts (T_C). Resolves ambiguous NL terms into the exact schema element they

refer to (e.g., “Apple” $\rightarrow$ brand.name rather than fruit.name).

(2) Value Grounding (T_V). Translates vague expressions or non-canonical mentions into concrete filter literals (e.g., “valid orders” $\rightarrow$ status=‘completed’; “NY” $\rightarrow$ ‘New York’).

(3) Join Path (T_P). Restores missing connections between distant entities, making multi-hop routes explicit (e.g., Student–Course bridged via the Enrollment table).

(4) Computation Template (T_F). Captures derived metrics as atomic operator subgraphs that serve as symbolic arithmetic templates, yielding an explicit computation trace and mitigating numerical hallucinations (e.g., “net profit” $\rightarrow$ revenue - cost).

(5) Logic Skeleton (T_L). Decomposes complex queries into step-by-step structures that delineate subquery and set-operation boundaries (e.g., “users who bought A but not B” $\rightarrow$ EXCEPT).

2.1.4 Construction of Domain Knowledge Resources

We organize $\mathcal{E}(G^{(i)})$ into two complementary repositories: an *augmented log* for instance-level retrieval, and a *symbolic knowledge base* for cross-instance consistency.

Augmented logs. For each historical sample $S_i = (x_i, y_i)$, we back-annotate the extracted knowledge onto its aligned structure:

$$D_i^{\text{aug}} = (x_i, y_i, G_{\text{align}}^{(i)}, \mathcal{E}(G^{(i)})), \quad (5)$$

yielding a few-shot library carrying both y_i and its aligned graph.

Symbolic knowledge base. We further distill standardized *Intent* $\rightarrow$ *Logic* primitives into a global library:

$$\mathcal{K}_{\text{base}} = \{k_j\}_{j=1}^M, \quad k_j = (I_{\text{intent}}^{(j)}, T_{\text{logic}}^{(j)}), \quad (6)$$

where $I_{\text{intent}}^{(j)}$ is a normalized NL description for dense retrieval and $T_{\text{logic}}^{(j)}$ is the corresponding structured prior (path template, logic skeleton, or computation graph).

Isomorphic patterns are merged, conflicts resolved by frequency-weighted majority vote, and the ANN index (FAISS) updated incrementally; $\mathcal{K}_{\text{base}}$ is then consumed by KODE-Solver.

2.2 KODE-Solver: SQL Generation Agent

KODE-Solver is a retrieval-augmented in-context learning engine (Wei et al., 2022) that generates executable SQL for a query q over schema D without

fine-tuning. It consumes the two resources produced by KODE-Miner— $\mathcal{K}_{\text{base}}$ and $\{D_i^{\text{aug}}\}$ —to assemble a knowledge-enhanced prompt (§2.2.1), and mitigates LLM stochasticity through consensus voting over sampled candidates (§2.2.2).

2.2.1 Knowledge-Enhanced Context Construction

We retrieve context via a stratified hybrid strategy in which deterministic rules take priority over semantic similarity:

Rule-based triggering matches explicit domain terms (e.g., “quarter-on-quarter growth”) against $\mathcal{K}_{\text{base}}$ and instantiates the corresponding SQL templates, enforcing hard business constraints. **Dense retrieval** recalls semantic interpretations (e.g., threshold mappings) and structural exemplars from $\{D_i^{\text{aug}}\}$ by topological similarity over G_{align} , rather than surface lexical overlap.

The retrieved components are serialized into a chain-of-thought prompt:

$$P = \Phi_{\text{tmpl}}(D' \oplus V \oplus E_{\text{struct}} \oplus R_{\text{know}} \oplus I_{\text{task}}), \quad (7)$$

combining the schema (D'), value hints (V), aligned exemplars (E_{struct}), domain rules (R_{know}), and task description (I_{task}).

2.2.2 Candidate Generation and Consensus Voting

Following self-consistency (Wang et al., 2023a), we sample N candidates $\{s_1, \dots, s_N\}$ from P and select one via a two-level clustering vote:

(1) **Execution clustering.** Candidates are grouped by execution-equivalence on the target DB; we retain the largest cluster $\mathcal{S}^*$.

(2) **Structural tie-breaking.** Within $\mathcal{S}^*$, we return the candidate whose G_s skeleton (operator-edge multiset) is the most frequent.

The selected SQL s^* and its cluster size $\text{Freq}_{\text{exec}}(s^*) = |\mathcal{S}^*|$ are passed to KODE-Critic.

2.3 KODE-Critic: Iterative Refinement Agent

KODE-Critic closes the loop by (i) verifying and surgically rewriting the candidate s^* against schema and evidence constraints (§2.3.1), and (ii) promoting high-confidence pairs for deployment-time knowledge accumulation (§2.3.2). We reuse the SQL-to-graph mapping from §2.1.1 to parse s^* into G_{s^*} , keeping all three agents on a consistent representation.

2.3.1 Graph-Guided Verification and Surgical Rewriting

We prompt an LLM to check G_{s^*} against the schema and the evidence graph G_q , collecting violations into a conflict set $\mathcal{C} = \mathcal{C}_{\text{sch}} \cup \mathcal{C}_{\text{sem}}$:

- $\mathcal{C}_{\text{sch}}$ (*schema validity*): non-existent tables/columns, type mismatches, and unreachable join paths.
- $\mathcal{C}_{\text{sem}}$ (*semantic alignment*): operator mismatches (e.g., SUM vs. AVG) and directional errors (e.g., flipped comparisons).

Each conflict c carries a severity weight w_c (schema errors outweigh semantic warnings), giving a scalar quality score:

$$\text{Score}_{\text{critic}}(s^*) = \frac{1}{1 + \sum_{c \in \mathcal{C}} w_c}. \quad (8)$$

If $\mathcal{C} \neq \emptyset$, **surgical rewriting** freezes nodes outside the conflict neighborhood and regenerates only the masked sub-structures (span-level infilling), iterating until $\mathcal{C} = \emptyset$ or a step budget is reached.

2.3.2 Confidence-Based Knowledge Backfill

A verified pair (q, s^*) is promoted to a high-confidence memory instance only when it survives both voting and verification:

$$\mathcal{D}_{\text{new}} = \left\{ (q, s^*) \mid \text{Freq}_{\text{exec}}(s^*)/N \geq \tau_{\text{cons}} \wedge \text{Score}_{\text{critic}}(s^*) \geq \tau_{\text{valid}} \right\}. \quad (9)$$

where $\tau_{\text{cons}} = 0.75$ and $\tau_{\text{valid}} = 1.0$. With $N = 8$, promotion therefore requires at least six execution-equivalent candidates and no residual conflict detected by the Critic after rewriting. These thresholds are fixed before the prequential evaluation. Promoted instances are fed back to KODE-Miner, which re-runs alignment (§2.1.2–2.1.3) and incrementally updates $\mathcal{K}_{\text{base}}$ and $\{D_i^{\text{aug}}\}$.

Each entry is versioned by source batch, and updates that conflict with the established majority in $\mathcal{K}_{\text{base}}$ are rolled back. This filtering reduces, but does not eliminate, erroneous promotions.

3 Experiments

We evaluate KODE-SQL under four complementary questions:

RQ1 (Domain Effectiveness & Portability): How does KODE-SQL perform on knowledge-heavy scenarios (ScienceBenchmark), and can

its mined knowledge seamlessly plug into other solvers?

RQ2 (Knowledge Source): Do gains come from structured implicit knowledge rather than SQL-example imitation?

RQ3 (Cross-Schema Transfer): Which abstract templates transfer to evaluation databases with unseen schemas?

RQ4 (Deployment Accumulation): Can confidence-filtered backfill improve future queries without future or within-batch leakage?

Additional analyses on data scaling, inference efficiency, and hyperparameter sensitivity are provided in Appendix C.

3.1 Experimental Setup

(1) Datasets. We evaluate KODE-SQL on three primary benchmarks: **BIRD** (Li et al., 2023), **Spider** (Yu et al., 2018), and **ScienceBenchmark** (Zhang et al., 2024) (detailed descriptions are provided in Appendix B.1).

(2) Comparison regimes. For ScienceBenchmark, BIRD, and Spider, KODE-Miner constructs memory exclusively from the corresponding official training split, and evaluation is performed on the validation/dev split; no evaluation question or gold SQL is used for mining, retrieval, or memory construction. ScienceBenchmark tests in-domain reuse, whereas BIRD and Spider use schema-disjoint train/dev databases to test abstract transfer. BIRD-Dev-Domain is the only derived setting: after a fixed, database-stratified random split, 460 BIRD-Dev examples (30%) are used as historical logs and the remaining 1,074 (70%) form $\mathcal{D}_{test}$, as detailed in §3.3. We distinguish *controlled comparisons* from heterogeneous literature results reported only for context in Appendix C.4.

(3) Evaluation Metrics. We employ Execution Accuracy (EX) to measure the correctness of query execution results against the ground truth.

(4) Implementation Details. For all controlled comparisons, we standardize the backbone, schema serialization, decoding and database-access budgets, and evaluator. Table 2 is a controlled full-system comparison that retains each method’s inference procedure; in paired experiments (Tables 3–5), the underlying solver pipeline is additionally fixed and only the stated KODE component varies. Full settings are in Appendix B.3.

(5) Statistical Reporting. Unless otherwise stated, EX results produced by our implementation are arithmetic means over eight independent

Table 2: **Framework Comparison on ScienceBenchmark.**

Method	Science Domains			Avg.
	CORDIS	ONCOMX	SDSS	
DIN-SQL	50.0	52.5	7.0	36.5
DAIL-SQL	50.0	50.1	10.0	36.7
MAC-SQL	51.0	55.6	14.0	40.2
PURPLE	54.0	51.5	24.0	43.2
CHESS	57.0	58.0	18.0	44.3
OpenSearch	56.0	60.6	19.0	45.2
DELLM	50.0	50.1	8.0	36.0
KAT-SQL	51.0	58.0	11.0	40.0
KODE-SQL	58.0	85.9	25.0	56.3

inference runs under the same configuration with independent candidate sampling. Across the repeated main comparisons and ablations, run-level sample standard deviations range from 0.2 to 2.6 EX points. Literature-reported results are reproduced as published and excluded from this aggregation; the prequential simulation uses one fixed shuffled order, and correction attribution uses one fixed 100-case sample.

3.2 Domain-Specific Effectiveness (RQ1)

We test KODE-SQL as a standalone framework and as a portable knowledge module on ScienceBenchmark; RQ2 then isolates the contribution of structured implicit knowledge.

3.2.1 Comparison on Science Benchmark

We compare all methods on ScienceBenchmark (CORDIS, ONCOMX, and SDSS) with the same gpt-4o-mini backbone, schema serialization, budgets, and evaluator, while retaining their method-specific inference procedures.

As shown in Table 2, KODE-SQL achieves the best average accuracy of **56.3%**, 11.1 points above OpenSearch (45.2%). The gain is largest on ONCOMX, where KODE-SQL reaches **85.9%**, compared with **60.6%** by OpenSearch.

3.2.2 Extensibility as a Plug-and-Play Module

Adding the mined knowledge to six Text-to-SQL solvers consistently yields **4.4–7.1 EX-point** gains (Table 3), demonstrating both standalone and plug-in effectiveness.

Table 3: **Plug-and-Play Evaluation on Science Benchmark.**

Method	Science Domains			Avg.	Gain
	CORDIS	ONCOMX	SDSS		
DIN-SQL	50.0	52.5	7.0	36.5	–
+ Ours	53.0	65.7	12.0	43.6	+7.1
DAIL-SQL	50.0	50.1	10.0	36.7	–
+ Ours	53.0	64.6	13.0	43.5	+6.8
MAC-SQL	51.0	55.6	14.0	40.2	–
+ Ours	54.0	69.7	15.0	46.2	+6.0
PURPLE	54.0	51.5	24.0	43.2	–
+ Ours	54.0	66.7	24.0	48.2	+5.0
CHESS	57.0	58.0	18.0	44.3	–
+ Ours	59.0	68.0	19.0	48.7	+4.4
OpenSearch	56.0	60.6	19.0	45.2	–
+ Ours	58.0	70.7	20.0	49.6	+4.4

3.3 Does Implicit Knowledge Drive the Gain? (RQ2)

To verify the effectiveness of Implicit Knowledge (IK) and simulate an enterprise scenario lacking human rules, we construct **BIRD-Dev-Domain**: the fixed split assigns 460 BIRD-Dev queries to historical logs ($\mathcal{D}_{history}$) and 1,074 to evaluation ($\mathcal{D}_{test}$). We enforce an **evidence-denied protocol**: human evidence is strictly excluded from both mining and inference (except in Exp 7 for reference). Table 4 reports ablations on Qwen2. 5-Coder-32B.

(1) **IK enriches demonstrations.** Adding IK to augmented demonstrations lifts EX from 40.1% to 46.6% (Exp 1 vs. 2, **+6.5 points**), as verbalized IK exposes the reusable logic that raw SQL logs leave opaque.

(2) **IK is a strong retrieval unit.** Augmenting an Explicit-only KB with IK yields a substantial **14.6-point** gain (Exp 3 vs. 4, 45.7% $\rightarrow$ 60.3%), indicating that IK serves as reusable domain logic. Combining both channels (Exp 5) reaches **64.1%**, confirming their complementarity.

(3) **Graph grounding prevents hallucination.** Removing graph constraints during mining (using only vanilla prompts) drops EX by **6.6 points** (Exp 5 vs. 6, 64.1% $\rightarrow$ 57.5%), showing that topological alignment is important for curbing unreliable LLM summarization.

(4) **Structured mined rules outperform unstructured human hints.** Under zero-shot settings, our Explicit+IK KB (Exp 4, 60.3%) outperforms the dataset’s original human hints (Exp 7, 45.9%) by **14.4 points**. This does not imply KODE-

Table 4: **Deep Dive on Structured Implicit Knowledge.** Evaluated on BIRD-Dev-Domain ($\mathcal{D}_{test}$). Human evidence is strictly excluded; Exp 7 uses original human hints solely as a reference.

ID	Mining Strategy	Demonstration	Retrieved KB	EX (%)
1	Miner (Align. only)	Augmented Logs	$\times$	40.1
2	Miner (Align. + IK)	Augmented Logs(+IK)	$\times$	46.6 (+6.5)
3	Miner (Align. only)	$\times$	Explicit KB	45.7
4	Miner (Align. + IK)	$\times$	Explicit + IK KB	60.3 (+14.6)
5	Miner (Align. + IK)	Augmented Logs(+IK)	Explicit + IK KB	64.1
6	w/o Graph (Align. + IK)	Augmented Logs(+IK)	Explicit + IK KB	57.5 (-6.6)
7	$\times$	$\times$	Human Hints ($\mathcal{D}_{test}$)	45.9

Table 5: Controlled schema-disjoint transfer. Values are EX (gain).

Method	BIRD-Dev	Spider-Dev
MAC-SQL	53.3	80.9
+ KODE-Xfer	56.3 (+3.0)	83.8 (+2.9)
CHESS	63.2	84.4
+ KODE-Xfer	66.4 (+3.2)	86.4 (+2.0)
OpenSearch	65.3	83.7
+ KODE-Xfer	67.5 (+2.2)	85.7 (+2.0)

SQL exceeds human intelligence, but rather that LLMs reason more reliably with strict, schema-grounded symbolic rules than with ambiguous natural-language hints.

In general, these results verify the effectiveness of implicit knowledge. By extracting and structuring such knowledge, KODE-SQL better captures hidden domain logic and helps bridge the gap between raw queries and complex SQL generation.

3.4 Schema-Disjoint Transfer (RQ3)

BIRD and Spider use disjoint training and evaluation schemas, so database-specific concepts (T_C), values (T_V), and concrete join paths (T_P) cannot be reused safely. We mine only from each official training split and transfer schema-agnostic computation templates (T_F), logic skeletons (T_L), and delexicalized typed topology patterns. During mining, source table, column, and literal identifiers are replaced by typed slots while operator and data-flow relations are retained. At inference, these slots are grounded exclusively against the target schema; neither source-schema identifiers nor source database contents are exposed. In every pair in Table 5, GPT-4o-mini, solver instructions, decoding, schema linker, budgets, and evaluator are fixed; only the abstract KODE block is added.

KODE-Xfer improves all six solver–benchmark pairs by 2.0–3.2 points. The transferred structures supply invariants that survive schema renaming: T_F preserves aggregation and arithmetic form, T_L preserves nesting and set-operation organization,

Table 6: **System-level ablations with Qwen2.5-Coder-32B-Instruct.** Parentheses show EX drops. The ScienceBenchmark setting differs from the GPT-4o-mini setting in Table 2.

Variant	BIRD-Dev	Spider-Dev	Science
KODE-SQL	64.3	85.0	58.0
w/o Miner	59.7 (-4.6)	80.8 (-4.2)	48.0 (-10.0)
w/o Solver	60.1 (-4.2)	81.5 (-3.5)	53.0 (-5.0)
w/o Critic	62.4 (-1.9)	82.9 (-2.1)	56.0 (-2.0)

Table 7: **Knowledge-base composition (%)**.

Type	BIRD-Domain	CORDIS	ONCOMX	SDSS
Knowledge	1,183	879	795	964
T_C	29.3	23.6	27.3	21.0
T_V	39.0	33.6	28.0	42.6
T_P	21.9	12.4	15.2	11.0
T_F	8.7	9.2	5.4	9.8
T_L	1.2	21.2	24.1	15.8

and typed topology preserves relation roles without copying a concrete path. Improvements across MAC-SQL, CHESS, and OpenSearch further indicate that this signal complements different generation pipelines rather than depending on one solver. The result also delineates what does not transfer: target-specific synonyms, default values, and concrete joins still require in-domain evidence. Heterogeneous full-system results are retained only as literature context in Appendix C.4.

3.5 Component and Knowledge Analysis

Table 6 isolates system components using Qwen2.5-Coder-32B-Instruct for all three datasets; it is a separate controlled setting from the GPT-4o-mini ScienceBenchmark comparison in Table 2. Thus, the 58.0 ScienceBenchmark score below and the 56.3 score in Table 2 are results from different backbones, not conflicting estimates under one configuration. Template ablations are in Appendix C.2. Removing Miner causes the largest loss, especially on ScienceBenchmark (-10.0), while removing Solver costs 3.5–5.0 points and removing Critic costs about two points. Leave-one-type-out results show complementary contributions from T_C, T_V, T_P, T_F (2.0–2.3 points each on BIRD-Dev-Domain); T_L has a smaller effect there because it constitutes only 1.2% of that knowledge base.

Correction attribution. We further trace corrected predictions to their supporting knowledge. On BIRD-Dev-Domain, KODE-SQL corrects 278 baseline failures while introducing 6 regressions.

Table 8: Attribution of 100 sampled corrections on BIRD-Dev-Domain. Counts are multi-label; “Any KODE” denotes at least one mined type.

Source	T_C	T_V	T_P	T_F	T_L	Other	Any KODE
# Cases	41	27	29	25	4	24	88

Table 9: Prequential deployment-time accumulation. P/C are query-level promotion precision/coverage (%). Gold execution results are used only for post-hoc measurement and the oracle, not for automatic promotion.

Setting	B1	B2	B3	B4	B5	P	C
Frozen empty KB	38.2	37.8	39.1	38.4	38.7	–	0.0
Eq. 9 backfill	38.2	47.6	54.9	56.8	60.4	92.1	47.6
Human-validated oracle	38.2	51.5	58.2	61.6	64.1	100	100

We manually multi-label 100 sampled corrections (Table 8); 88 invoke at least one mined type, while 24 also involve non-knowledge factors such as candidate selection or Critic rewriting. T_C appears most often (41 cases), and T_V, T_P , and T_F each support 25–29 cases. Because labels are not mutually exclusive, these counts identify contributing mechanisms rather than partitioning the total. Together with the leave-one-type-out results in Appendix C.2, this analysis attributes most sampled corrections to mined knowledge without claiming that every gain comes from the knowledge base alone.

3.6 Deployment-Time Accumulation (RQ4)

We simulate cold-start deployment on BIRD-Dev-Domain with a prequential protocol. We randomly shuffle $\mathcal{D}_{test}$ and partition it into five equal-sized sequential batches. Each batch is evaluated with a frozen KB, and only completed batches may update later ones, preventing future and within-batch leakage. We compare a frozen empty KB, automatic Eq. 9 backfill with the fixed thresholds specified in §2.3.2, and a human-validated oracle upper bound. In the oracle condition, every completed query is validated and, when necessary, corrected to a gold-equivalent SQL before mining; no current- or future-batch item is exposed early.

We measure promotion precision as the number of automatically promoted query–SQL pairs whose execution result matches the gold result, divided by all automatically promoted pairs. Coverage is the fraction of eligible processed queries that produce a promoted pair. Gold results are consulted only to compute these two post-hoc statistics; the automatic policy itself observes only execution consensus and Critic conflicts. Automatic back-

Table 10: Fixed-history sample efficiency on BIRD-Dev-Domain. The 0% setting contains no mined knowledge.

Log ratio	0%	10%	25%	50%	100%
EX	38.2	51.7	58.4	61.9	64.1

fill improves final-batch EX by 21.7 points over the frozen KB and recovers 85.4% of the human-validated oracle improvement, with 92.1% promotion precision at 47.6% coverage. This supports confidence-filtered deployment-time accumulation. Section 3.7 separately measures sample efficiency when a fixed quantity of validated history is available.

3.7 Fixed-History Sample Efficiency and Cost

We also vary the amount of validated history available before evaluation, independently of the online protocol above. On BIRD-Dev-Domain, the same randomly sampled fractions of $\mathcal{D}_{history}$ are used to build a fixed KB for every test query.

Table 10 shows that 10% and 25% of the logs recover 52.1% and 78.0% of the full 25.9-point improvement over the empty-KB setting. Performance then rises more gradually to 64.1, suggesting that frequent concepts and join patterns are captured early while rarer rules require broader coverage.

Additional sensitivity, efficiency, and convergence analyses are provided in Appendix C. We use $k = n = 4$ as a fixed default; the post-hoc sensitivity analysis was not used for model selection. On BIRD-Dev, increasing k to 16 with $n = 4$ lowers EX from 64.3 to 61.2, while $k = n = 16$ yields 59.8, showing that selective retrieval is preferable to adding more historical context.

KODE-SQL uses 2.5K tokens and 8.6 seconds per query, compared with 3.9K/12.5 seconds for CHESS and 3.7K/11.2 seconds for OpenSearch. Relative to CHESS, this reduces tokens by 35.9% and latency by 31.2%, with graph construction and retrieval adding under 0.5 seconds. On BIRD-Dev, the Critic improves EX from 62.4 to 64.3 within two rounds; Round 3 brings no further gain, supporting the three-round stopping cap.

4 Conclusion

We presented **KODE-SQL**, which distills unmatched regions of historical (NL, SQL) pairs into five reusable template types. Its Miner aligns dual-view graphs, its Solver combines rule-dense retrieval with consensus, and its Critic verifies candi-

dates and filters memory updates.

In-domain experiments show gains of up to **11.1 EX points** and consistent plug-in improvements across six solvers. Controlled ablations attribute the gains to structured implicit knowledge rather than surface SQL imitation. Under schema-disjoint evaluation, abstract KODE templates yield smaller but consistent 2.0–3.2 point gains; under prequential deployment, confidence-filtered backfill improves future-batch EX while maintaining 92.1% promotion precision.

The evidence therefore supports KODE-SQL as a domain-oriented knowledge layer for deployments with available or gradually accumulating traces, rather than a universal zero-log replacement for specialized Text-to-SQL systems.

Limitations

KODE-SQL has five main boundaries.

Historical data. KODE targets deployments with validated in-domain traces; with an empty KB it falls back to the base solver. The fixed-history and prequential studies show early gains, but 47.6% promotion coverage leaves many processed queries unable to expand the KB.

LLM verification. Using an LLM for both generation and judgment risks confirmation bias. Consensus and graph checks reduce this risk, but 92.1% promotion precision still permits contamination; high-stakes use requires cross-model, symbolic, or human validation.

Cross-schema transfer. Concepts, values, and concrete join paths remain schema-specific. Only computations, logic skeletons, and delexicalized topology transfer safely in our setting, with observed gains of 2.0–3.2 points.

Scope and cost. We evaluate three scientific databases and two general benchmarks, but not continual schema drift, additional dialects, adversarial logs, or very long-lived KBs. Online latency also excludes one-time mining and recurring audit, deduplication, and retirement costs.

Risks. Systematic errors or bias in historical SQL can be distilled and reapplied. The Critic detects graph conflicts but cannot establish that a legacy convention is semantically sound; sensitive deployments therefore require periodic human audits that track template provenance and retire entries after schema or policy changes.

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- 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*, pages 26569–26583. Association for Computational Linguistics.
- Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. [Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources](#). In *Proceedings of the 2018 International Conference on Management of Data*.
- Asim Biswal, Chuan Lei, Xiao Qin, Aodong Li, Balakrishnan Narayanaswamy, and Tim Kraska. 2026. [Agentsm: Semantic memory for agentic text-to-sql](#). *arXiv preprint arXiv:2601.15709*.
- Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. [Rsl-sql: Robust schema linking in text-to-sql generation](#). *arXiv e-prints*.
- Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2025. [Sqlfixagent: Towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration](#). In *AAAI*.
- Shuaichen Chang and Eric Fosler-Lussier. 2023. [How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings](#). *arXiv e-prints*, arXiv:2305.11853.
- Shuaichen Chang and Eric Fosler-Lussier. 2023. [Selective demonstrations for cross-domain text-to-SQL](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14174–14189. Association for Computational Linguistics.
- Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. [M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, Bangkok, Thailand. Association for Computational Linguistics.
- Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2025. [Reliable text-to-sql with adaptive abstention](#). *Proceedings of the ACM on Management of Data*.
- Zhibo Chu, Zichong Wang, and Qitao Qin. 2024. [Leveraging Prior Experience: An Expandable Auxiliary Knowledge Base for Text-to-SQL](#). *arXiv e-prints*, arXiv:2411.13244.
- Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. [C3: Zero-shot Text-to-SQL with ChatGPT](#). *arXiv e-prints*, arXiv:2307.07306.
- Longxu Dou, Yan Gao, Xuqi Liu, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, Min-Yen Kan, and Jian-Guang Lou. 2022. [Towards knowledge-intensive text-to-sql semantic parsing with formulaic knowledge](#). In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5240–5253.
- Jonathan Fürst, Catherine Kosten, Farhad Nooralahzadeh, Yi Zhang, and Kurt Stockinger. 2024. [Evaluating the Data Model Robustness of Text-to-SQL Systems Based on Real User Queries](#). *arXiv e-prints*, arXiv:2402.08349.
- 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*.
- Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. [Few-shot text-to-sql translation using structure and content prompt learning](#). *Proceedings of the ACM on Management of Data*.
- Zijin Hong, Zheng Yuan, Hao Chen, Qinggang Zhang, Feiran Huang, and Xiao Huang. 2024. [Knowledge-to-sql: Enhancing sql generation with data expert llm](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10997–11008.
- Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. [Next-generation database interfaces: A survey of llm-based text-to-sql](#). *IEEE Transactions on Knowledge and Data Engineering*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. [Qwen2.5-Coder Technical Report](#). *arXiv e-prints*, arXiv:2409.12186.
- Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. [Billion-scale similarity search with GPUs](#). *IEEE Transactions on Big Data*, 7(3):535–547.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. [Large language models are zero-shot reasoners](#). *Advances in neural information processing systems*, 35:22199–22213.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. [Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353.

- 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).
- Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. [The dawn of natural language to SQL: Are we fully ready?](#) *Proceedings of the VLDB Endowment*, 17(11):3318–3331.
- Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. [Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search](#). *arXiv e-prints*, arXiv:2502.17248.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. [Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs](#). *arXiv e-prints*, arXiv:2305.03111.
- Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Hangyu Mao, and Rui Zhao. 2025. [Pet-sql: A prompt-enhanced two-round refinement of text-to-sql with cross-consistency](#). In *International Conference on Database Systems for Advanced Applications*, pages 193–208. Springer.
- 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 Transactions on Knowledge and Data Engineering*, 37(10):5735–5754.
- Xiping Liu and Zhao Tan. 2023. [Divide and Prompt: Chain of Thought Prompting for Text-to-SQL](#). *arXiv e-prints*, arXiv:2304.11556.
- Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026a. [XiYan-SQL: A novel multi-generator framework for text-to-SQL](#). *IEEE Transactions on Knowledge and Data Engineering*, pages 1–14.
- Yu Liu, Wenxiao Zhang, Diandian Guo, Cong Cao, Fangfang Yuan, Qiang Sun, Yanbing Liu, Jin B. Hong, and Zhiyuan Ma. 2026b. [Does faithfulness-guided alignment hurt accuracy? unlocking accurate and faithful post-retrieval reasoning](#). *arXiv preprint arXiv:2602.01348*.
- Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. [Natural language to sql: State of the art and open problems](#). *Proceedings of the VLDB Endowment*, 18(12):5466–5471.
- 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](#). In *NeurIPS 2024 Third Table Representation Learning Workshop*.
- Yu A. Malkov and Dmitry A. Yashunin. 2018. [Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs](#). *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836.
- Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. [Enhancing text-to-sql capabilities of large language models: A study on prompt design strategies](#). In *Findings of the association for computational linguistics: EMNLP 2023*, pages 14935–14956.
- OpenAI. 2024. [GPT-4o system card](#). *arXiv preprint arXiv:2410.21276*.
- You Peng, Youhe Jiang, Wenqi Jiang, Chen Wang, and Binhang Yuan. 2026. [Hexgen-flow: Optimizing llm inference request scheduling for agentic text-to-sql](#). *2026 IEEE 42nd International Conference on Data Engineering (ICDE)*, pages 1759–1772.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Taleai, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. [Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql](#). *International Conference on Learning Representations*, 2025.
- Mohammadreza Pourreza and Davood Rafiei. 2023. [Din-sql: Decomposed in-context learning of text-to-sql with self-correction](#). *Advances in Neural Information Processing Systems*, 36:36339–36348.
- Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. [Evaluating the Text-to-SQL Capabilities of Large Language Models](#). *arXiv e-prints*, arXiv:2204.00498.
- Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X Sean Wang. 2024. [Purple: Making a large language model a better sql writer](#). In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 15–28. IEEE.
- Tonghui Ren, Chen Ke, Yuankai Fan, Yinan Jing, Zhenying He, Kai Zhang, and X Sean Wang. 2025. [The power of constraints in natural language to sql translation](#). *Proceedings of the VLDB Endowment*, 18(7):2097–2111.
- Lei Sheng and Xu Shuai Shuai. 2025. [Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning](#). In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 1473–1496.
- Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. [A survey on employing large language models for text-to-sql tasks](#). *ACM Computing Surveys*, 58(2):1–37.

- Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. [Exploring chain of thought style prompting for text-to-SQL](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5376–5393, Singapore. Association for Computational Linguistics.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. [CHES: Contextual Harnessing for Efficient SQL Synthesis](#). *arXiv e-prints*, arXiv:2405.16755.
- Blerta Veseli, Simon Razniewski, Jan-Christoph Kalo, and Gerhard Weikum. 2023. [Evaluating the knowledge base completion potential of GPT](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6432–6443, Singapore. Association for Computational Linguistics.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Li Zhoujun. 2025. [Mac-sql: A multi-agent collaborative framework for text-to-sql](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557.
- Liangwei Wang, Zhengxuan Zhang, Yi-Fan Cao, Fugee Tsung, and Yuyu Luo. 2026. [Tabletale: Reviving the narrative interplay between data tables and text in scientific papers](#). In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1–17.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. [Self-consistency improves chain of thought reasoning in language models](#). In *The Eleventh International Conference on Learning Representations*.
- Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. [Self-instruct: Aligning language models with self-generated instructions](#). In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 13484–13508.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). *Advances in neural information processing systems*, 35:24824–24837.
- Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. [Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment](#). *Proceedings of the ACM on Management of Data*, 3(3):1–24.
- Yuanzhen Xie, Xinzhou Jin, Tao Xie, Matrixmxlin Matrixmxlin, Liang Chen, Chenyun Yu, Cheng Lei, Chengxiang Zhuo, Bo Hu, and Zang Li. 2024. [Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10796–10816.
- Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. [Wizardlm: Empowering large pre-trained language models to follow complex instructions](#). In *International Conference on Learning Representations*, volume 2024, pages 30745–30766.
- Chenyu Yang, Yuyu Luo, Chuanxuan Cui, Ju Fan, Chengliang Chai, and Nan Tang. 2025. [Data imputation with limited data redundancy using data lakes](#). *Proceedings of the VLDB Endowment*, 18(10):3354–3367.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. [Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921. Association for Computational Linguistics.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025a. [AFlow: Automating agentic workflow generation](#). In *The Thirteenth International Conference on Learning Representations*.
- Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. 2025b. [Structure-guided large language models for text-to-sql generation](#). In *Proceedings of the 42nd International Conference on Machine Learning, ICML’25*. JMLR.org.
- Tingkai Zhang, Chaoyu Chen, Cong Liao, Jun Wang, Xudong Zhao, Hang Yu, Jianchao Wang, Jianguo Li, and Wenhui Shi. 2024. [SQLfuse: Enhancing Text-to-SQL Performance through Comprehensive LLM Synergy](#). *arXiv e-prints*, arXiv:2407.14568.
- Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2024. [Sciencebenchmark: A complex real-world benchmark for evaluating natural language to SQL systems](#). *Proceedings of the VLDB Endowment*, 17(4):685–698.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2022. [Least-to-Most Prompting Enables Complex Reasoning in Large Language Models](#). *arXiv e-prints*, arXiv:2205.10625.
- Yizhang Zhu, Shiyin Du, Boyan Li, Yuyu Luo, and Nan Tang. 2024. [Are large language models good statisticians?](#) *Advances in Neural Information Processing Systems*, 37:62697–62731.

A Related Work

The advent of Large Language Models (LLMs) has fundamentally transformed the landscape of Text-to-SQL parsing (Rajkumar et al., 2022; Luo et al., 2025). Different from traditional supervised learning approaches (Hong et al., 2025; Liu et al., 2025), our work generally focuses on In-Context Learning (ICL) and Hybrid Frameworks. Current research generally unleashes the potential of LLMs through prompt engineering, reasoning enhancement, and external knowledge integration. In this section, we review the existing literature from three primary perspectives: reasoning enhancement strategies, execution-guided refinement, and knowledge-augmented generation.

A.1 Reasoning Enhancement and Prompting Strategies

For accurate translation of complex queries, enhancing the intrinsic reasoning process of LLMs is crucial. The Chain-of-Thought (CoT) prompting (Wei et al., 2022; Kojima et al., 2022) serves as a foundational technique, encouraging models to generate intermediate reasoning steps akin to logical entailment. Building upon this, various strategies have been proposed to optimize the prompt design (Chang and Fosler-Lussier, 2023; Nan et al., 2023). For instance, (Liu and Tan, 2023) and (Tai et al., 2023) explore improved CoT styles specifically for SQL generation. Additionally, selecting high-quality demonstrations—based on skill similarity or semantic relevance (Gu et al., 2023; Shi et al., 2025)—can significantly boost performance. To further improve robustness and mitigate randomness, Self-consistency (Talei et al., 2024; Wang et al., 2023a) and multi-path aggregation methods (Lee et al., 2025; Dong et al., 2023) have been introduced, which sample multiple reasoning paths and select the most frequent answer via majority voting. Beyond outcome accuracy, CRAFT studies post-retrieval multi-hop question answering and jointly rewards answer correctness, citation validity, and semantic faithfulness to produce more auditable reasoning traces under noisy evidence (Liu et al., 2026b). We cite it as broader evidence-faithful reasoning work, rather than as a Text-to-SQL method.

A.2 Task Decomposition and Multi-Agent Frameworks

To surmount the limitations of single-pass reasoning in complex scenarios, current methodolo-

gies prioritize task decomposition and collaborative agent workflows. Seminal works like Least-to-Most prompting (Zhou et al., 2022) and DIN-SQL (Pourreza and Rafiei, 2023) break down query generation into manageable sequential sub-tasks, a paradigm that has expanded into specialized multi-agent frameworks like MAC-SQL (Wang et al., 2025) and SQLFixAgent (Cen et al., 2025) which distribute generation and verification roles. This process is further fortified by iterative self-correction mechanisms, incorporating internal critique (Ren et al., 2025; Li et al., 2026), execution feedback (Biswal et al., 2026), and adaptive abstention (Chen et al., 2025) to ensure robustness.

A.3 Knowledge Augmentation and Synthetic Data

To mitigate LLMs’ limitations in handling domain-specific nuances and large-scale schemas, early works integrated external knowledge via formulaic banks (Dou et al., 2022) or manual benchmarks (Li et al., 2023), though these approaches remain labor-intensive. Consequently, research has pivoted towards automated data generation to tackle scarcity, utilizing synthetic question-SQL pairs (Chang and Fosler-Lussier, 2023; Hong et al., 2024), large-scale instruction tuning datasets (Wang et al., 2023b; Xu et al., 2024), and historical query logs (Chu et al., 2024; Pourreza et al., 2025). However, given the potential noise in synthetic data and the schema-agnostic nature of general knowledge bases (Veseli et al., 2023), current efforts focus on constructing reliable, automated in-domain knowledge bases to support robust retrieval-augmented generation (Zhang et al., 2024; Liu et al., 2026a).

Distinction from retrieval and agent memory.

These approaches and KODE-SQL can be separated along four dimensions. *Memory unit*: demonstration retrieval stores a whole NL–SQL example, while execution or semantic memory stores an interaction-level experience or feedback record (Chu et al., 2024; Biswal et al., 2026); KODE stores a normalized Intent→Logic subgraph template and keeps the original trace only as a secondary exemplar. *Construction*: prior experience is recorded or retrieved as a successful case, whereas KODE aligns the NL and SQL views and explicitly extracts SQL regions lacking NL grounding. *Use*: case memory primarily supplies in-context precedents or correction evidence, whereas KODE first instantiates exact symbolic rules and only then

retrieves structurally similar exemplars. *Schema scope*: KODE marks concepts, values, and concrete join paths as schema-bound, while permitting only delexicalized computation, logic, and typed-topology templates to transfer across schemas. The contribution is therefore not a larger cache of historical SQL, but a decomposition of each trace into explicit evidence, reusable implicit rules, and schema-scoped exemplars.

B Experimental Setup and Implementation Details

This section provides comprehensive details regarding our experimental design. We first introduce the diverse datasets used for evaluation, followed by a categorization of the baseline methods. Finally, we specify the hyperparameter settings to ensure the reproducibility of our framework.

B.1 Dataset Descriptions

Our evaluation uses BIRD and Spider for database-disjoint generalization and ScienceBenchmark for in-domain reuse. For official benchmark evaluations, we retain the original authors’ splits and execution evaluators. BIRD-Dev-Domain is the only derived setting and is constructed from BIRD-Dev as described in §3.3.

- **BIRD** (Li et al., 2023): A large-scale, cross-domain benchmark containing 12,751 question–SQL pairs over 95 databases totaling 33.4 GB. In addition to complex SQL, it emphasizes noisy values, content grounding, and external evidence needed for conventions, calculations, and joins not revealed by schema names. Transfer templates are mined only from the official training databases; source identifiers and contents remain hidden during evaluation.
- **Spider** (Yu et al., 2018): A standard cross-domain benchmark with 10,181 questions and 5,693 distinct SQL queries over 200 databases. Its database-level split tests compositional SQL, multi-table linking, nesting, aggregation, ordering, and set operations without memorized schemas. We use it to evaluate delexicalized template transfer while withholding database-specific concepts and values.
- **ScienceBenchmark** (Zhang et al., 2024): An existing benchmark introduced by Zhang et

al., comprising three complex, real-world scientific databases (CORDIS, OncoMX, and SDSS) and manually created as well as automatically generated NL/SQL pairs.

B.2 Baseline Methods

To empirically validate the effectiveness of KODE-SQL, we conduct a comprehensive comparison against a wide array of state-of-the-art (SOTA) Text-to-SQL methods. The selected baselines span the most representative lines of recent research, including: C3-SQL (Dong et al., 2023), DIN-SQL (Pourreza and Rafiei, 2023), DAIL-SQL (Gao et al., 2024), SuperSQL (Li et al., 2024), MCS-SQL (Lee et al., 2025), Distillery (Maa-mari et al., 2024), RSL-SQL (Cao et al., 2024), CHESS-SQL (Talaie et al., 2024), XiYan-SQL (Liu et al., 2026a), PURPLE (Ren et al., 2024), DELLM (Hong et al., 2024), MAC-SQL (Wang et al., 2025), SGU-SQL (Zhang et al., 2025b), OpenSearch-SQL (Xie et al., 2025), CHASE-SQL (Pourreza et al., 2025), Alpha-SQL (Li et al., 2025), CSC-SQL (Sheng and Shuai, 2025) and KAT-SQL (Baek et al., 2025). We group them into four categories based on their core methodological focus, as detailed below.

Prompting-based Strategies.

- **C3-SQL**: A zero-shot method emphasizing clear context, calibration, and consistent output formatting.
- **DIN-SQL**: A decomposition-based method breaking complex queries into sub-tasks (schema linking, join determination, etc.).
- **DAIL-SQL**: A strong few-shot baseline that selects diverse and semantically similar examples for in-context learning.

Agentic & Multi-Stage Frameworks.

- **MAC-SQL**: A multi-agent framework utilizing a refiner agent to correct SQL queries based on execution feedback.
- **CHESS-SQL**: An interactive framework combining information retrieval with a verify-and-refine pipeline for large-scale schemas.
- **XiYan-SQL / CHASE-SQL**: Recent competitive frameworks optimizing the reasoning chain specifically for large foundation models.

Knowledge-Enhanced Methods.

- **DELLM**: A framework that retrieves and injects auxiliary definitions to assist generation.
- **KAT-SQL**: A retrieval-augmented generation approach integrating external knowledge bases.
- **PURPLE**: Focuses on logical operator composition and structural similarity for retrieving demonstrations.

B.3 Implementation Details

Controlled tables pair variants with identical inference configurations and vary only the stated KODE component. Appendix Table 13 aggregates heterogeneous published and reproduced systems for context, not fairness claims.

LLM Backbones. We evaluate four proprietary and open-source backbones to test cross-model generality:

- **GPT-4o** (OpenAI, 2024): used as the strongest proprietary backbone for upper-bound comparison.
- **GPT-4o-mini** (OpenAI, 2024): a lightweight model in the same GPT-4o family, used for cost-accuracy trade-off analysis.
- **Qwen2.5-Coder-14B-Instruct** and **Qwen2.5-Coder-32B-Instruct** (Hui et al., 2024): open-source code-specialized backbones, used to verify that our gains do not depend on a single model family.

Each controlled KODE-baseline pair uses the same backbone.

Retrieval Module. **BGE-M3** (Chen et al., 2024) encodes NL queries and SQL graph nodes; **FAISS** (Johnson et al., 2019) with HNSW (Malkov and Yashunin, 2018) performs similarity search:

- Number of retrieved few-shot demonstrations: $n = 4$.
- Number of retrieved knowledge items per query: $k = 4$.
- Knowledge-filter cosine threshold: $\tau = 0.75$ (entries below this threshold are discarded to suppress noisy retrievals).

Decoding and Voting. To control output variance:

- Temperature is set to **0** for all single-pass generation (deterministic decoding).
- For self-consistency / majority voting, we sample $N = 8$ candidates with temperature $T = 0.7$ and top- $p = 0.95$, then apply the two-level execution-clustering vote described in §2.2.2.
- Maximum context length: 8,192 tokens for open-source backbones; default window sizes (128K) for proprietary APIs.

Repeated Runs and Variability. For each repeated experiment, candidate generation, execution-based selection, and evaluation are rerun independently under one configuration. We report arithmetic means over eight runs and the sample standard deviation of run-level EX. Benchmark splits stay fixed; only stochastic inference varies.

Hardware and Runtime. All experiments are conducted on a server with $8 \times$ NVIDIA A100 (80GB) GPUs. Open-source backbones are served via vLLM; proprietary models are accessed through their official APIs with rate-limit retries.

Controlled-Comparison Protocol. Table 2 is a controlled full-system comparison: all methods are reproduced with the same backbone LLM, database schema serialization, decoding and database-access budgets, and evaluator, while retaining their method-specific inference procedures.

For Tables 3, 4, and 5, each paired comparison additionally keeps the underlying solver pipeline fixed and varies only the stated KODE component. The same number of in-context demonstrations ($n = 4$) is used where applicable.

These controls do not apply to heterogeneous literature-reported systems in Table 13.

B.4 Scientific Artifacts: Licenses, Intended Use, and Documentation

B.4.1 Licenses and Terms of Use

All artifacts used in this work are publicly released under research-permissive licenses, and our usage strictly complies with their original terms.

Datasets. **BIRD** (Li et al., 2023) uses CC BY-NC 4.0 for non-commercial research; **Spider** (Yu

et al., 2018) uses CC BY-SA 4.0; and **ScienceBenchmark** (Zhang et al., 2024) uses its authors’ research-only license. We access all three from their official repositories and preserve the original test splits.

Backbone Models. **GPT-4** and **GPT-4o-mini** are accessed via OpenAI’s official API in compliance with OpenAI’s Terms of Use. **Qwen2.5-Coder-14B** and **Qwen2.5-Coder-32B** (Hui et al., 2024) are released under the Apache 2.0 license and used through their official open-source weights.

Baselines. All baseline methods compared in Table 13 are either reported from their original publications or reproduced using publicly released code under their respective open-source licenses (typically MIT or Apache 2.0). We did not redistribute any baseline code or weights.

B.4.2 Intended Use

We use BIRD, Spider, and ScienceBenchmark solely for non-commercial Text-to-SQL evaluation and report derived results without redistributing the datasets. KODE-SQL and its mined templates are intended for research on knowledge-augmented semantic parsing and inherit the access conditions of the underlying benchmarks.

B.4.3 Personally Identifying Information and Offensive Content

The datasets used in this work (BIRD, Spider, ScienceBenchmark) are public Text-to-SQL benchmarks that contain database schemas, natural-language questions about those schemas, and the corresponding SQL queries. They do not contain personally identifying information (PII) of real individuals, nor do they include offensive or harmful content. We did not collect any new data involving human participants, and therefore no anonymization procedure was required. The mined knowledge templates in our knowledge base are abstracted from schema-level patterns (e.g., join paths, computation formulas) and likewise contain no PII.

B.4.4 Documentation of Artifacts

The three benchmarks cover a diverse set of English-language domains and SQL phenomena: BIRD spans 95 databases across 37 domains (e.g., finance, sports, healthcare) and emphasizes real-world value grounding and external knowledge; Spider covers 200 cross-domain databases focused on compositional generalization; ScienceBenchmark targets three scientific domains—Research

Policy Making (CORDIS), Astrophysics (SDSS), and Cancer Research (OncoMX)—and emphasizes domain-specific terminology and knowledge-intensive queries. All three benchmarks are in English. Detailed domain coverage and characteristics are described in Section 3 (Experimental Setup).

B.4.5 Dataset Statistics

For the official BIRD and Spider evaluations, we use the train/dev/test splits released by the original benchmark authors. ScienceBenchmark uses its domain-specific training resources and development sets. BIRD-Dev-Domain is the only derived setting and is constructed from BIRD-Dev as described in §3.3. The exact statistics are:

- **BIRD:** 9,428 training, 1,534 development, and 1,789 test examples over 95 databases.
- **Spider:** 8,659 training, 1,034 development, and 2,147 test examples over 200 databases.
- **ScienceBenchmark:** manual-seed/synthetic-training/ development counts are 100/1,306/100 (CORDIS), 100/2,061/100 (SDSS), and 100/1,065/100 (OncoMX).
- **BIRD-Dev-Domain:** 460 historical-log and 1,074 evaluation examples from the fixed 30%/70% split.

C Supplementary Experiments and Results

In this section, we present extended experimental results that complement the main text.

C.1 Computational Setup and Implementation Packages

Backbone Models. We evaluate KODE-SQL with both proprietary backbones (GPT-4, GPT-4o-mini) accessed via the official OpenAI API, and open-source backbones (Qwen2.5-14B-Instruct, Qwen2.5-32B-Instruct) used directly from their official released weights. No training or fine-tuning is performed in this work; all experiments are inference-only.

Hyperparameters. LLM decoding uses temperature = 0 for deterministic generation. The Solver’s stratified hybrid retrieval returns top- $k = 4$ entries per query from $\mathcal{K}_{\text{base}}$. The Critic allows up to $R = 3$ revision rounds before terminating. P4 receives the evaluator’s target `sql_dialect`; we do not globally rewrite queries into SQLite.

Implementation Packages. Our implementation is in Python and uses the following publicly available packages with default settings: **PyTorch** and **Transformers** for model inference; **vLLM** for batched serving of Qwen2.5 backbones; **sentence-transformers** and **FAISS** for dense retrieval over the knowledge base; **sqlglot** for SQL parsing and AST normalization used by KODE-Critic; and **NetworkX** for constructing the Semantic Program Graph. Execution Accuracy is computed using the official evaluation scripts released by the BIRD, Spider, and ScienceBenchmark authors, without modification.

C.2 Component and Knowledge-Type Analysis

Table 11 reports the leave-one-type-out effects complementing the main-text composition analysis.

Table 11: **Leave-one-type-out ablation** on BIRD-Dev-Domain.

Setting	EX	Δ
Full KODE-SQL	64.1	–
w/o T_C	62.1	-2.0
w/o T_V	61.8	-2.3
w/o T_P	61.9	-2.2
w/o T_F	62.0	-2.1
w/o T_L	63.8	-0.3

C.3 Retrieval Parameter Sensitivity

We investigate the interplay between retrieved knowledge items (k) and few-shot demonstrations (n). All main experiments use the prespecified default $k = n = 4$. Table 12 is a post-hoc sensitivity analysis on BIRD-Dev and was not used for model selection.

Sweet Spot. Performance does not increase linearly. The fixed configuration also has the highest observed accuracy in this sweep (64.3%), consistent with selective knowledge and concise demonstrations limiting context noise.

Diminishing Returns and Context Noise. Deviating from the peak reveals two key observations:

- **Knowledge Saturation:** Increasing k from 4 to 16 while fixing $n = 4$ causes a drop from 64.3% to 61.2%. This confirms that excessive schema rules can act as noise and distract the LLM from the core intent.
- **Context Overflow:** The most resource-intensive setting ($k = 16, n = 16$) yields

sub-optimal results (59.8%), even lower than using no knowledge at all ($k = 0, n = 4$). We attribute this to the *Lost-in-the-Middle* phenomenon, where relevant cues are buried in an overly lengthy prompt.

Table 12: Sensitivity Analysis of Knowledge (k) and Demonstrations (n). We report Execution Accuracy (%) on BIRD-Dev.

k / n	0	4	8	16
0	56.2	60.1	60.8	60.5
4	59.8	64.3	64.2	63.1
8	59.4	63.8	63.5	62.0
16	57.5	61.2	60.9	59.8

C.4 Detailed Experimental Results

We provide the complete performance comparison of KODE-SQL against an extensive list of state-of-the-art baselines on BIRD and Spider benchmarks in Table 13.

C.5 Convergence of KODE-Critic

We further track the refinement stability of KODE-Critic. By leveraging topological checking to mask verified SQL skeletons, the LLM only edits localized errors. As shown in Table 14, execution accuracy converges rapidly from 62.4% at Round 0 to 64.3% within two rounds. The number of newly introduced errors remains near zero, showing that KODE-Critic can fix structural errors without causing unstable error accumulation.

C.6 Efficiency and Cost Analysis

We analyze KODE-SQL’s complexity and inference cost on BIRD-Dev-Domain with Qwen2.5-Coder-32B.

Complexity and Cost. KODE-Miner aligns compact query-level semantic graphs rather than the entire global schema, making its $O(|V|^2)$ bipartite matching cost negligible even for large schemas. As shown in Table 15, KODE-SQL requires 2.5K total tokens and 8.6s latency per query, compared with 3.9K tokens and 12.5s for CHESS. The auxiliary overhead, including retrieval and graph construction, remains lightweight ($< 0.5s$).

D Implementation Details

D.1 SQL Operator Specifications

To ensure the rigorous mapping of natural language intent to database logic, KODE-Miner utilizes a de-

Table 13: **Comprehensive Main Results.** Full SOTA comparison on BIRD and Spider. Best results are in **bold**; “–” indicates unreported results.

Method + Backbone	BIRD	Spider	
	Dev	Dev	Test
<i>Foundation Models</i>			
GPT-4	48.3	68.6	71.1
GPT-4o-mini	46.9	68.4	73.2
Qwen2.5-14B	48.1	67.7	72.1
Qwen2.5-32B	47.9	70.8	77.9
<i>Closed-Source Backbones</i>			
C3-SQL (GPT-3.5)	50.2	81.8	82.3
DIN-SQL (GPT-4)	50.7	82.8	85.3
DAIL-SQL (GPT-4)	55.9	84.4	86.6
MAC-SQL (GPT-4)	57.5	86.8	82.8
PURPLE (GPT-4)	–	86.7	87.8
DELLM (GPT-4)	39.4	71.8	–
SuperSQL (GPT-4)	58.5	86.6	87.0
Distillery (GPT-4o)	67.2	–	–
RSL-SQL (GPT-4o)	67.2	–	87.9
MCS-SQL (GPT-4)	64.4	89.5	89.6
SGU-SQL (GPT-4)	61.8	87.9	–
CHESS-SQL (GPT-4)	68.3	–	87.2
OpenSearch (GPT-4o)	69.3	–	87.1
XiYan-SQL	73.3	–	89.7
CHASE (Gemini-1.5)	73.0	–	87.6
<i>Open-Source Backbones</i>			
CHESS-SQL (Qwen-32B)	63.1	–	–
Alpha-SQL (Qwen-14B)	67.8	87.0	–
Alpha-SQL (Qwen-32B)	69.7	–	–
CSC + XiYan (Qwen-32B)	71.3	–	–
KAT-SQL (Llama-3-70B)	41.3	74.6	–
KAT-SQL (Granite-20B)	57.5	–	–
<i>KODE-SQL (Ours)</i>			
w/ GPT-4	65.9	86.5	87.0
w/ GPT-4o-mini	64.1	84.5	84.3
w/ Qwen2.5-14B	63.1	83.1	83.3
w/ Qwen2.5-32B	64.3	85.0	86.3

Table 14: **Convergence analysis of KODE-Critic on BIRD-Dev.**

Setting	Round 0	Round 1	Round 2	Round 3
Execution Accuracy	62.4%	63.9%	64.3%	64.3%
Corrected Errors	–	28	9	1
Newly Introduced Errors	–	5	1	1

finer set of atomic SQL operators within its graph representation (G_s). Table 16 outlines the semantic mapping between standard SQL clauses and our normalized operator set, along with the protocol specifications used to enforce validity during graph construction.

Table 15: **Efficiency comparison on BIRD-Dev-Domain.** “Aux. Time” is the non-LLM overhead, including retrieval and validation.

Method	#Calls	In Tok.	Out Tok.	Total Tok.	Aux. Time	Latency
CHESS	3.5	3.5K	0.4K	3.9K	<0.5s	12.5s
OpenSearch	2.8	3.2K	0.5K	3.7K	0.8s	11.2s
KODE-SQL	2.1	2.2K	0.3K	2.5K	<0.5s	8.6s

Normalization procedure. We traverse the parsed SQL abstract syntax tree from data sources to outputs and instantiate one node per semantic operation. Schema elements and literals remain typed slots, while directed edges record data flow and slot filling. Within each query block,

SCAN introduces relations; JOIN and FILTER restrict them; GROUPBY and AGG define aggregation scope; CALC represents typed scalar expressions and casts; and PROJECT, ORDERBY, and TOPK form the output. Nested queries are normalized recursively and attached through NEST, while SETOP combines schema-compatible query blocks. The result abstracts away clause formatting while retaining dependencies needed for alignment and localized rewriting; aliases resolve within scope, whereas typed literals and semantics-sensitive operator order are preserved.

Validity contract. Before alignment, graph construction checks schema membership, operand and aggregation type compatibility, join reachability, operator scope, and structural constraints associated with individual SQL operators. For example, HAVING must follow aggregation, may reference only grouping keys or aggregated expressions, and cannot expose an ungrouped free column. Similarly, join edges must correspond to reachable and type-compatible schema paths. A violation is retained as an explicit, localized schema conflict for KODE-Critic rather than being silently discarded or automatically rewritten. These checks establish graph well-formedness but do not guarantee execution equivalence or semantic correctness; whether a structurally valid operator expresses the user’s intent remains a semantic-alignment decision handled jointly by the Miner and Critic.

Alignment interface. The normalized graph is the SQL-side input to the partial alignment A^* in §2.1.2. Type compatibility first limits which language-evidence nodes may ground an operator slot. Each SQL node then retains its alignment state $\mathcal{M}(u)$: grounded nodes become explicit anchors, whereas ungrounded nodes remain visible as candidates for implicit knowledge. We deliberately preserve, rather than delete, the latter nodes. Their data-flow neighbors provide the context needed to distinguish a missing lexical match from genuinely unstated logic, such as a bridge join, derived calculation, or default filter. Connected unmatched regions are subsequently passed to template extraction; isolated SQL tokens are never treated as reusable domain rules without their operator context. The same operator identities let KODE-Critic attach conflicts to affected nodes and freeze verified regions during localized rewriting.

Table 16: **Specification of SQL Operators.** Mappings and validity constraints used by KODE-Miner.

Operator	SQL	Protocol Specification
SCAN	FROM	Executes base table access and pushes predicates down to the Filter operator.
PROJECT	SELECT	Defines output schema via attribute selection and deduplication (DISTINCT).
FILTER	WHERE	Reduces cardinality through logical predicates without altering the relation schema.
JOIN	JOIN	Merges schemas from dual execution plans subject to join type and key compatibility.
GROUPBY	GROUP BY	Restructures data granularity and enforces an aggregatable validity scope on keys.
AGG	AGG	Derives scalar metrics via aggregation functions (SUM, AVG, etc.) applied to compatible types.
CALC	arithmetic/CAST	Represents typed scalar expressions and casts, enforcing operand and result-type compatibility.
HAVING	HAVING	Applies filtering at the subgroup level, prohibiting unaggregated free variables.
ORDERBY	ORDER BY	Imposes tuple ordering without changing the logical composition of the set.
TOPK	LIMIT	Enforces a hard cardinality limit based on implementation defaults.
NEST	EXISTS/IN	Resolves existential or set-based quantification using correlated subqueries.
SETOP	UNION/ INTERSECT/ EXCEPT	Combines query blocks with equal arity and compatible output types.

D.2 Prompt Templates

This appendix lists the five core prompts used in KODE-SQL. For brevity, schema fragments, in-context demonstrations, and long JSON snippets are abbreviated as “...”; runtime placeholders are shown in **magenta**. All prompts are issued via in-context learning without fine-tuning. Table 17 summarizes how the five prompts map to the agents and pipeline stages.

Every placeholder is populated from the typed artifacts produced by the preceding stage: schema slots expose only valid identifiers, graph slots preserve node and edge types, and retrieved items retain template provenance. Agent outputs are parsed against the displayed JSON or SQL contract before entering the next stage; malformed outputs are retried within the existing call budget rather than

silently accepted. The abbreviated prompts therefore omit long instances, not protocol constraints.

Table 17: Mapping between prompts and pipeline stages.

ID	Prompt	Stage
P1	NL Graph Construction	Miner §2.1.1
P2	NL–SQL Alignment (multi-turn)	Miner §2.1.2
P3	Knowledge Mining	Miner §2.1.3
P4	Knowledge-Guided SQL Generation	Solver §2.2.1
P5	Graph-Guided Verification & Rewriting	Critic §2.3.1

Prompt handoff contract. Together, P1–P5 pass typed artifacts from query graph and alignment through template extraction and candidate generation to localized verification.

D.2.1 P1 – NL Graph Construction

Given a user query q , this prompt parses q into the Language Evidence Graph G_q (§2.1.1), producing typed entity/operator nodes and attribution/modification edges. The output is later consumed by P2 as one side of the alignment.

Prompt P1 – NL Query Graph Construction

Role

You are an NL Semantic Parser. Convert a user query into a typed evidence graph $G_q = (V_q, E_q)$ that preserves linguistic evidence rather than executable logic.

Node Types

- **Query Object:** domain target (e.g., “movie”, “director”).
- **Constant:** temporal / numeric / value literal (e.g., “2023”, “top 10”).
- **Qualifier:** descriptive or categorical condition (e.g., “action”, “Chinese”).
- **Operator:** operational intent (e.g., “filter”, “exclude”, “rank”).

Edge Types

- **attr:** belonging / composition between objects.
- **mod:** a constant or qualifier modifying an object.

Inputs

User query: **{{query}}**

Schema context (table/column names only): **{{schema}}**

Output (JSON)

```
{
  "nodes": [ { "id", "type", "text", "span": [s,e],
  ... ],
  "edges": [ { "src", "dst", "label", ... },
  "reasoning": "brief rationale"
}
```

Rules

1. Keep uncertain modifiers; do not force executable precision.
2. Resolve pronouns before modifying (“their courses” → owner entity).
3. Each node **must** carry a character span from the original query.

D.2.2 P2 – NL–SQL Alignment (Multi-turn)

This prompt implements dual-view alignment (§2.1.2) as a multi-turn loop. Each turn refines the accumulated history $\mathcal{H}_t$ with one typed alignment layer.

Prompt P2 – NL–SQL Alignment (Multi-turn, System Prompt)

Role

Align NL evidence with the Semantic Program Graph G_{align} over multiple turns. Extend or refine history $\mathcal{H}_t$; never restart from scratch.

Turn Schedule

- **T1 Skeleton:** align normalized operator nodes to NL spans; otherwise mark them implicit.
- **T2 Entities:** align tables, columns, and literals (including N:1/1:N); mark unstated required schema items as implicit_concept.
- **T3 Calculations:** align each canonical formula or mark it derived/intermediate.
- **T4 Joins:** align paths, keys, intent, and usage; flag bridge tables required for reachability.

Cross-turn Rules

1. **Monotonicity:** only refine or extend confirmed alignments from $\mathcal{H}_{t-1}$.
2. **Residue:** revisit every unmatched NL span and SQL node from turn $t-1$.
3. **Context:** describe each implicit item in this query, not as a generic definition.

Per-turn Output (JSON)

```
{ "turn": t,
  "alignments": [ { "sql_node",
  "nl_span", "kind", "confidence", "reason",
  ... },
  "residue": { "nl_unmatched": [ ... ],
  "sql_unmatched": [ ... ] },
  "next_tool": "entity|calc|join|done" }
```

Inputs Per Turn

History / SQL AST: **{{H_t}}** / **{{sql_{ast}}}**

NL graph / schema: **{{G_q}}** / **{{schema}}**

Current calc/join subgraph: **{{subgraph}}**

Alignment Validity

1. A match must be type-compatible and supported by a span or a previously confirmed alignment.
2. One NL span may license several SQL nodes only when the relation is explicitly recorded as one-to-many.
3. Never convert an unmatched SQL node into a match merely to exhaust the residue; retain it as implicit evidence.
4. Preserve node identifiers across turns so that later updates refine, rather than duplicate, earlier decisions.

Failure Handling

If required context is missing, return the unresolved node in residue with a reason and request the corresponding tool input; do not fabricate a schema or semantic match.

Termination

Emit done when sql_unmatched is empty or unchanged for two turns. Example turn: ...

Confidence

Score each alignment independently; do not raise confidence merely because the residual set becomes smaller.

D.2.3 P3 – Knowledge Mining

Given a fully aligned Semantic Program Graph G_{align} , this prompt extracts the five template families $\{T_C, T_V, T_P, T_F, T_L\}$ defined in §2.1.3: concept mapping, value grounding, join path, computation template, and logic skeleton.

Prompt P3 – Knowledge Mining (Five Templates)

Role

You are the Knowledge Miner. Given the aligned graph G_{align} , extract **reusable, domain-level priors** as five template families.

Templates to Extract

- T_C **Concept Mapping**: ambiguous NL term $\rightarrow$ exact schema element.
- T_V **Value Grounding**: vague NL value $\rightarrow$ canonical literal (e.g., “valid orders” $\rightarrow$ status=‘completed’).
- T_P **Join Path**: ordered table-key chain bridging distant entities, with intent and usage scenario.
- T_F **Computation Template**: derived metric as an operator subgraph (name, standard_form, dependencies, dependency_sources).
- T_L **Logic Skeleton**: step-by-step abstract pattern with subquery / set-op boundaries (schema-agnostic, parameterized).

Sources to Use

1. **Implicit subgraphs** (nodes with $\mathcal{M}(u) = \varepsilon$) are the primary mining targets.
2. **Residual NL evidence** R_q provides candidate naming and intent.
3. **Domain schema** disambiguates concept references.

Inputs

G_{align} : **{{Galign}}**

Residual NL evidence R_q : **{{Rq}}**

Calculation/Join subgraphs (if any): **{{subgraphs}}**

Schema: **{{schema}}**

Output (JSON, keyed by template family)

```
{
  "T_C": [{"nl_term", "schema_target",
    "scope", "confidence"}, ... ],
  "T_V": [{"nl_phrase", "column",
    "literal", "explanation"}, ... ],
  "T_P": [{"path": [{"table", "key"}],
    "intent", "scenario"}, ... ],
  "T_F": [{"name", "standard_form", "deps",
    "dep_sources", "explanation"}, ... ],
  "T_L": [{"pattern_name",
    "steps": [...], "category"}, ... ],
  "reasoning": "why each item is reusable
    beyond this instance"
}
```

Rules

1. Every template item **must** be backed by at least one implicit node in G_{align} or one residual NL span.
2. Strip instance-specific literals from T_L ; keep them in T_V .
3. Skip pass-through aliases that are not real computations.
4. Normalize identifiers into typed slots while preserving operator order, scope, and provenance.

D.2.4 P4 – Knowledge-Guided SQL Generation

The Solver prompt (§2.2.1) assembles the retrieved priors into a chain-of-thought generation instruction. Compared with conventional schema-only prompts, it injects three additional channels: deterministic rule hits, structural exemplars retrieved by G_q similarity, and value-grounding hints.

Prompt P4 – Knowledge-Guided SQL Generation

Role

You are a precise **{{sql_dialect}}** query writer. Use the supplied domain knowledge as **authoritative priors** – prefer them over your own guesses whenever they apply.

Schema D'

{{schema}}

Value Hints V (column $\rightarrow$ sampled literals)

{{value_hints}}

Triggered Rules R_{know} (must-apply)

{{rules}}

Each rule has the form “NL pattern $\Rightarrow$ required SQL fragment”. Apply every triggered rule unless it conflicts with the explicit user request.

Aligned Exemplars E_{struct} (for structure, not content)

{{exemplars}}

Borrow the operator skeleton, not the literals.

User Task I_{task}

{{query}}

Reasoning Procedure

1. **Intent restatement**: rewrite the question in SQL-aware form (what to SELECT, FILTER, GROUP, ORDER).
2. **Knowledge check**: list which rules / exemplars / value hints apply, and why.
3. **Skeleton draft**: write the operator skeleton (SELECT / FROM / JOIN / WHERE / GROUP / ORDER / LIMIT).
4. **Grounding**: fill in concrete columns and literals using the value hints.
5. **Self-check**: verify each WHERE condition has a source in either the user query or a triggered rule.

Output Format

```
#reasoning: ...
#applied_knowledge: [rule_ids, exemplar_ids]
#sql:
“sql
...
“
```

Constraints

- Preserve the target dialect’s numeric semantics; introduce casts only when required by the intended operation.
- Output exactly one final SQL block; no commentary after the SQL.

Knowledge and Validation

Explicit intent outranks stored knowledge; rules precede exemplars, which supply structure only. Verify identifiers, joins, aggregation, projections, and predicate provenance; repair failures before emitting the final SQL block.

D.2.5 P5 – Graph-Guided Verification and Rewriting

The Critic prompt (§2.3.1) parses the candidate s^* into G_{s^*} , checks it against the schema and the NL evidence G_q , and – when conflicts exist – performs **surgical rewriting** that regenerates only the masked sub-structures while the conflict-free skeleton stays frozen.

Prompt P5 – Graph-Guided Verification & Surgical Rewriting

Role

You are the SQL Critic. Your job is to (i) audit the candidate s^* over its graph G_{s^*} , and (ii) repair conflicts **without disturbing** the parts that are already correct.

Inputs

Candidate SQL: $\{\{s_star\}\}$

Candidate graph G_{s^*} : $\{\{Gs_star\}\}$

NL evidence graph G_q : $\{\{Gq\}\}$

Schema + foreign keys: $\{\{schema\}\}$

Applied knowledge from Solver: $\{\{applied_knowledge\}\}$

Step A – Conflict Detection

Classify each violation into:

- C_{sch} : non-existent table/column, type mismatch, unreachable join path.
- C_{sem} : operator mismatch (SUM vs. AVG), flipped comparison, missing filter mandated by a triggered rule, redundant join.

For each conflict, return: {node_id, type, severity, evidence, suggested_fix}.

Step B – Surgical Rewriting (only if $C \neq \emptyset$)

1. **Freeze** all G_{s^*} nodes *outside* the 1-hop neighborhood of any conflict node; these form the skeleton.
2. **Mask** the conflict nodes and their immediate dependencies.
3. **Regenerate only the masked region**; emit the revised SQL together with a diff explaining what changed and why.
4. **Do not touch** frozen nodes; if a fix requires changing a frozen node, escalate by marking "escalate":true instead.

Output (JSON)

```
{
  "conflicts": [...],
  "action": "accept|rewrite|escalate",
  "revised_sql": "... or null",
  "diff": "... or null"
}
```

Stopping

Return "action":"accept" when $C = \emptyset$. Otherwise iterate at most $\{\{max_steps\}\}$ times; if still non-empty, escalate.