SQLAgent: Learning to Explore Before Generating as a Data Engineer

Wenjia Jiang[1], Yiwei Wang[2], Boyan Han[1], Joey Tianyi Zhou[3,4], Chi Zhang[1]*
[1]AGI Lab, Westlake University, China [2]University of California, Merced, USA
[3]CFAR, Agency for Science, Technology and Research, Singapore
[4]IHPC, Agency for Science, Technology and Research, Singapore
https://github.com/Westlake-AGI-Lab/SQLAgent

Abstract

Large Language Models have recently shown impressive capabilities in reasoning and code generation, making them promising tools for natural language interfaces to relational databases. However, existing approaches often fail to generalize in complex, real-world settings due to the highly database-specific nature of SQL reasoning, which requires deep familiarity with unique schemas, ambiguous semantics, and intricate join paths. To address this challenge, we introduce a novel two-stage LLM-based framework that decouples knowledge acquisition from query generation. In the Exploration Stage, the system autonomously constructs a database-specific knowledge base by navigating the schema with a Monte Carlo Tree Search–inspired strategy, generating triplets of schema fragments, executable queries, and natural language descriptions as usage examples. In the Deployment Stage, a dual-agent system leverages the collected knowledge as in-context examples to iteratively retrieve relevant information and generate accurate SQL queries in response to user questions. This design enables the agent to proactively familiarize itself with unseen databases and handle complex, multi-step reasoning. Extensive experiments on large-scale benchmarks demonstrate that our approach significantly improves accuracy over strong baselines, highlighting its effectiveness and generalizability.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in complex reasoning tasks(OpenAI, 2023; Bubeck et al., 2023; Mirchandani et al., 2023; Wu et al., 2023; Meta Fundamental AI Research (FAIR) Diplomacy Team et al., 2022). As high-value information resides primarily in relational databases(Verbitski et al., 2017; Yavuz et al., 2018), leveraging LLMs to interact with structured data has gained significant attention. This interest drives Text-to-SQL research, which translates natural language questions into executable SQL queries to democratize data access(Liu et al., 2025; Katsogiannis-Meimarakis and Koutrika, 2023; Kobayashi et al., 2025; Malekpour et al., 2024; Shi et al., 2025; Deng et al., 2022). The primary objective involves bridging the gap between human language and databases without requiring users to master complex syntax.

However, current approaches struggle to generalize to complex scenarios(Lei et al., 2025). Performance often drops on real-world databases and reveals a significant generalization gap(Pourreza and Rafiei, 2024; Gao et al., 2024; Talaei et al., 2024). This limitation arises because SQL reasoning is inherently database-specific rather than portable. Valid queries depend strictly on unique schema structures unlike generic code generation. Consequently, models face challenges in navigating intricate schemas and resolving semantic ambiguity based on specific column names. Furthermore, reasoning requires precise alignment with unique database join paths and relationships.

Without prior familiarity with the database, a generic pre-trained model struggles to navigate the unique structure and meaning within a new database(Lei et al., 2025). In contrast, human experts succeed by first building a deep familiarity with the database's unique schema and relationships. This core insight motivates our approach: a preliminary process designed to build this foundational knowledge before attempting the final translation task. To achieve this goal, we propose a novel LLM-based agent framework that operates in two key stages. First, the framework autonomously explores an unfamiliar database to generate a rich, database-specific knowledge base. Subsequently, this knowledge is provided to the agent as in-context examples(Dong et al., 2024;

*Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 39311–39331

Rubin et al., 2022; Zhang et al., 2022b). This process guides the generation of the final, complex SQL query.

The Exploration Stage autonomously constructs a structured, database-specific knowledge base. The core objective of this stage is to generate a rich set of usage examples for each key component of the database schema, such as its tables and columns. Each example is formalized as a triplet: a schema sub-structure, a corresponding executable SQL query, and its natural language description. To systematically generate these triplets across the entire schema, we first represent the database as a traversable tree structure, where entities like tables and columns are organized as nodes. This tree provides a map for our exploration. To generate triplets, the agent need explore this tree to find and combine meaningful nodes into valid queries. To this end, We then employ a search strategy inspired by Monte Carlo Tree Search(Kocsis and Szepesvári, 2006; Coulom, 2007), guided by an Agent that acts as the core reasoning engine. The agent intelligently navigates this tree to build and test new queries. It achieves this by selecting a series of actions, such as introducing a join or adding a filter. Each successful exploration path results in a new triplet. These triplets are collected to build our knowledge base, enabling the next stage to better write accurate SQL queries. Overall, this process allows the system to proactively familiarize itself with an unknown database without any manual intervention.

In the Deployment Stage, our goal is to effectively utilize the knowledge from the exploration phase to handle complex user queries. To achieve this, we introduce a dual-agent framework where an InfoAgent and a GenAgent collaborate with distinct roles. The InfoAgent grounds the user query in the database schema, identifying the most relevant tables and columns through semantic search over pre-computed column embeddings and LLM-driven context expansion to infer implicit join keys. The GenAgent then retrieves the most semantically relevant (S, Q, U) triplets from the knowledge base and uses them as database-specific in-context examples to guide the final SQL generation. If the generated query is unsuccessful, execution feedback is passed back to the InfoAgent, which prunes the schema context and retrieves additional candidates, prompting a new generation cycle. The two agents thus collaborate in an iterative loop, continuously refining the SQL query through a cycle of context refinement, example retrieval, generation, and execution feedback. This collaborative, multi-step design allows our system to deconstruct the problem, leading to high accuracy and reliability.

To validate the effectiveness of our approach, we conducted extensive experiments showing that our method significantly outperforms strong baselines on complex, large-scale benchmarks. Our main contributions are:

1. We propose a novel two-stage LLM-based framework that decouples knowledge acquisition from query generation.

2. We develop an autonomous, agent-driven exploration strategy that constructs a structured, executable knowledge base without requiring manual annotations.

3. We introduce a dual-agent reasoning mechanism that iteratively retrieves and integrates in-context examples to generate accurate and executable SQL queries.

2 Related Work

LLM-based Text-to-SQL Methods. Research in Text-to-SQL has progressed from early neural parsers to modern LLM-driven approaches (Shi et al., 2025; Deng et al., 2022). Seminal neural methods introduced techniques like graph-based encoders to leverage database schema (Wang et al., 2020) and constrained decoding to ensure syntactic correctness (Scholak et al., 2021). With the advent of large language models, the field has seen significant advancements. Numerous fine-tuning methods (Li et al., 2024) and advanced LLM-prompting techniques (Dong et al., 2023; Wang et al., 2023a; Zhang et al., 2023; Talaei et al., 2024; Pourreza and Rafiei, 2024; Gao et al., 2024) have achieved strong performance on established benchmarks. However, these approaches still contend with challenges such as limited context windows and adapting to unseen database domains.

Multi-Agent for Code Generation. The intersection of generative models and interactive problem-solving has spurred a surge in agent-based frameworks designed to enhance the reasoning capabilities of language models, particularly for code generation (Yao et al., 2023; Zhang et al., 2022a; Chen et al., 2023; Wang et al., 2023b; Shinn et al., 2024; Zhang et al., 2024; Xia et al., 2024). To overcome the limits of a single model, a prominent

strategy is to deploy multiple agents in coordinated roles. In this multi-agent paradigm, agents specialize and interact, often through patterns like cooperative decomposition (Li et al., 2023a) or hierarchical task delegation (langgenius, 2023). To make these interactions more robust, other works have focused on designing special action spaces to standardize agent operations (Wang et al., 2024; Yang et al., 2024). Inspired by these successes, our method employs a similar strategy with dedicated explorer and generator agents for text-to-SQL translation.

3 Methodology

The core insight behind our approach is that an agent familiar with a database generates SQL far more reliably than one reasoning over raw schema definitions alone. We therefore propose a two-stage framework to generalize Text-to-SQL to unfamiliar enterprise databases. An **Exploration Stage** autonomously constructs a database-specific knowledge base before any user query arrives. A subsequent **Deployment Stage** employs a dual-agent framework to leverage this knowledge for accurate SQL synthesis. We first introduce the compact schema representation that underpins both stages in §4.1, then detail the exploration process in §4.2, and finally describe the deployment mechanism in §4.3.

3.1 Database Schema Representation

Large-scale enterprise databases commonly exhibit structural redundancy. Time-based sharding produces hundreds of tables with identical schemas, such as daily logs or hourly snapshots. Modeling each table independently would attach every table node to its own set of field nodes, yielding $O(N \times M)$ edges for N sharded tables and M shared fields, making the schema intractably large for LLM-guided traversal.

To address this, we represent the relational schema as a traversable, tree-like graph in which databases, tables, and fields are distinct nodes connected by typed edges. The key abstraction within this structure is the **Shared Field Group**. Formally, let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be the set of tables in a database. Each table T_i is characterized by its schema signature $S_i = \{(f_{i,j}, \tau_{i,j})\}_{j=1}^{m}$, where $f_{i,j}$ and $\tau_{i,j}$ denote the name and data type of the j-th field, respectively. We define a **Shared Field Group** $\mathcal{G}_k = \{T_i \in \mathcal{T} \mid \text{Hash}(\text{sort}(S_i)) = \text{Hash}(\text{sort}(S_k))\}$, where $\text{sort}(\cdot)$ renders the signature invariant to column ordering and $\text{Hash}(\cdot)$ produces a unique 128-bits identifier for each structural pattern. Each table then establishes a single link to its corresponding Shared Field Group rather than individual connections to all its field nodes. This reduces edge complexity from $O(N \times M)$ to $O(N + M)$, streamlines LLM-guided traversal, and improves the interpretability of inter-table relationships. The identification algorithm is detailed in Appendix C and Appendix D.

3.2 Exploration Stage: LLM-Guided Tree Search

The Exploration Stage aims to systematically acquire prior knowledge about the target database before query time. Concretely, we populate a knowledge base with triplets (S, Q, U), where S is a schema sub-structure, Q is an executable SQL query over that sub-structure, and U is the aligned natural language description. During deployment, these triplets serve as database-specific in-context examples that ground query generation in verified schema patterns.

Generating diverse, valid triplets across a schema with hundreds of tables and thousands of columns is combinatorially challenging. A tree-structured search offers a principled solution by incrementally expanding schema nodes and directing the exploration budget toward productive regions. Monte Carlo Tree Search (Kocsis and Szepesvári, 2006) provides the structural inspiration for this design. Standard MCTS, however, relies on scalar rewards propagated via the Upper Confidence Bound for Trees. This assumption does not hold for SQL reasoning: query validity is binary at execution time, while the semantic quality of a query reflects how meaningfully it exercises the schema's relational structure, a property that is continuous and context-dependent. We therefore replace the UCT policy with an **LLM-as-Policy-and-Evaluator** paradigm. The LLM encodes error traces and schema semantics to emit rich natural-language feedback that distinguishes syntactically invalid paths from semantically unproductive ones, providing a more informative guidance signal than sparse numerical rewards.

The exploration continues until a predefined termination condition is met, namely a target number of valid triplets or a maximum iteration budget. The process consists of four phases, illustrated in Figure 1.

LLM-Guided Selection and Expansion. At

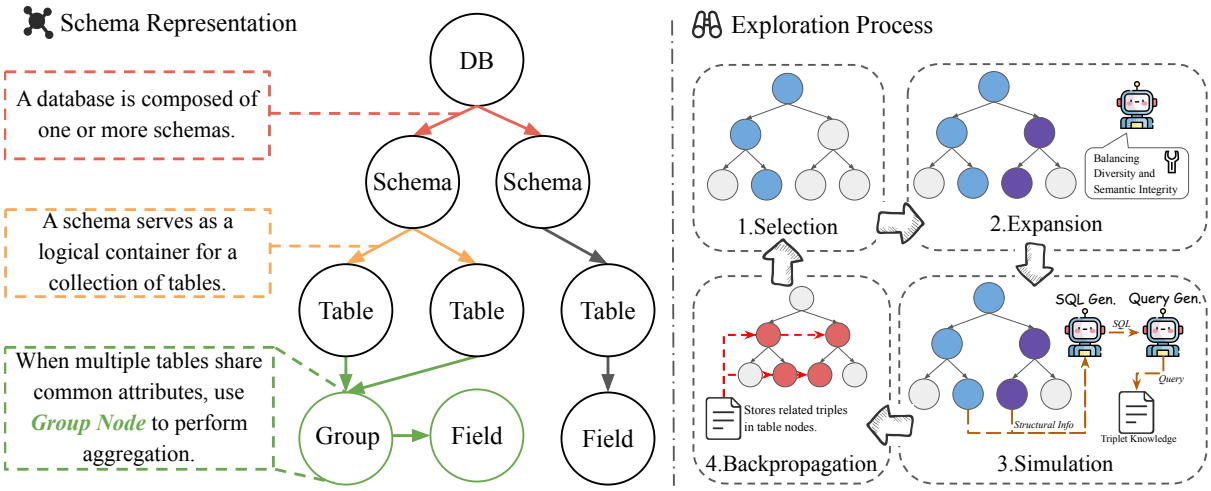


Figure 1: **Database Representation and Exploration Stage.** The left panel illustrates our schema representation: the **Shared Field Group** consolidates structurally identical tables into a single canonical node, reducing schema complexity from $O(N \times M)$ to $O(N + M)$. The right panel depicts the LLM-guided MCTS-inspired tree search, which traverses this compact representation through four phases, namely selection, expansion, simulation, and backpropagation, to accumulate a structured knowledge base of (S, Q, U) triplets.

each node in the search tree, the LLM receives the current query state, which is a sequence of prior actions, and the available schema context, which is a simplified JSON representation of reachable tables and columns. Its task is to select the most promising next action from a discrete action space that includes options such as `Select Column`, `Add Constraint`, and `Apply Aggregation`. This selection directly creates a new child node, expanding the tree toward more complex and semantically diverse query structures. This single reasoned step combines selection and expansion, ensuring broad coverage of database operations and schema interactions. A full description of the action space is provided in Appendix E.

SQL Simulation. Each expanded node represents a candidate schema sub-structure whose practical utility can only be confirmed through execution against the live database. From this node, an LLM generates a complete SQL query whose structure is semantically consistent with the current schema context. To promote meaningful exploration, the model is instructed to prioritize columns with associated documentation and to restrict selection to schema nodes not yet incorporated in the current path, retaining join-key columns as an exception to preserve relational connectivity. As a concrete example, if the node contains `age`, `height`, and `gender` from a `users` table, the LLM might generate `SELECT * FROM users WHERE age > 20 AND gender = 'Male'` along with a natural language description stating that the query retrieves all male users older than 20.

The resulting query undergoes a two-stage validation. First, a SQL parser checks syntactic correctness. Second, the query is executed against the live database to verify runtime validity. A simulation succeeds only if the query is syntactically valid, executes without runtime errors, and returns a non-empty, non-trivial result set. On success, an LLM generates the natural language component U from S and Q, completing the triplet. Because each example is grounded in a real schema sub-structure and validated against live data, this approach guarantees high semantic alignment among S, Q, and U.

Backpropagation. Without memory of prior attempts, the search would revisit unproductive schema paths repeatedly, exhausting the exploration budget on regions already proven fruitless. We address this by propagating outcomes back along the search path to guide future iterations. For a successful triplet (S, Q, U), positive feedback is recorded on all entity nodes, that is, the tables and columns that contributed to the query, enriching each schema component with a history of productive usage. For failed explorations, negative feedback lowers the selection priority of that path, discouraging repetition of unproductive combinations. In subsequent selection steps, the LLM prompt is augmented with this accumulated node-level feedback, creating a self-refining exploration process that progressively steers the search toward valid, semantically rich queries. Once the knowledge base

is sufficiently populated, the system transitions to the Deployment Stage, which draws on this accumulated knowledge to ground query generation in verified schema experience.

3.3 Deployment Stage: Dual-Agent SQL Synthesis

To bridge the semantic gap between a user's natural language query and a complex, unfamiliar database schema, we propose a dual-agent framework illustrated in Figure 2. The **InfoAgent** handles schema grounding and context management; the **GenAgent** performs knowledge-driven SQL synthesis. The two agents collaborate through an iterative refinement loop in which execution feedback drives progressive context pruning and query improvement.

InfoAgent: Schema Grounding & Context Management. A user query rarely names database columns directly, and naive keyword matching over a schema with hundreds of tables frequently misses critical join keys or misidentifies relevant columns. To address this, the InfoAgent constructs a precise, executable schema context in two steps.

The first step is *Schema Grounding*. To make the schema semantically searchable, each column is serialized into a descriptive string by concatenating its name, data type, and any available documentation comment. This string is then encoded into a high-dimensional vector using a sentence-embedding model, producing a pre-computed embedding index over the entire schema. At query time, the InfoAgent extracts semantic keywords from the user query and retrieves the top-k most relevant columns by performing a similarity search against this index.

The second step is *Context Expansion*. The initial top-k set is often incomplete, as it surfaces the directly relevant columns while omitting the join keys necessary to link them into an executable query. The InfoAgent therefore prompts an LLM to act as a database expert, providing it with both the user query and the preliminary schema set. The LLM reasons about the relational dependencies among the retrieved components and infers any logically necessary additions. For instance, if the initial retrieval surfaces a `user_name` column from a `users` table alongside an `order_amount` column from an `orders` table, the LLM identifies and adds the `user_id` and `customer_id` keys required to join the two tables. The resulting schema context, which contains both directly retrieved and logically inferred components, is then transmitted to the GenAgent.

GenAgent: Knowledge-Driven Synthesis. Even with an accurate schema context, generating complex SQL for an unfamiliar database is difficult without concrete usage examples that demonstrate how the schema components interact. The GenAgent addresses this by grounding generation in the triplet knowledge base built during the Exploration Stage. It generates a retrieval embedding by jointly encoding the user's natural language question and the InfoAgent-provided schema context through a code-embedding model. Encoding the question alone would surface triplets that match its intent but may involve unrelated schema structures. By incorporating the schema context into the embedding, retrieval is directed toward examples that are both semantically relevant and structurally compatible with the target database. A similarity search is then performed against the pre-computed embeddings of the SQL component Q in each stored triplet, retrieving the top-k triplets whose query vectors are most semantically similar to the current request. These triplets, which encode verified schema patterns alongside their natural language descriptions, serve as database-specific few-shot examples. The GenAgent then constructs a final prompt by combining the user query, the refined schema context, and these retrieved examples to produce the SQL query. Example prompts and agent interaction details for both agents are given in Appendix B.

Collaborative Refinement Loop. A single-pass generation is rarely sufficient for complex enterprise queries, where an incomplete schema context or an undetected join dependency can invalidate the entire result. We address this with a collaborative refinement loop in which the two agents iteratively narrow the schema context and correct the query until execution succeeds. After the GenAgent produces a query, it is executed against the live database. If execution fails, the InfoAgent responds with two concurrent corrective actions. First, it **prunes** the schema context by removing components that were present in the context but absent from the generated query, thereby narrowing the candidate set and reducing noise for the next iteration. Second, it re-analyzes the user query for secondary semantic keywords that the initial grounding step may have overlooked, using them

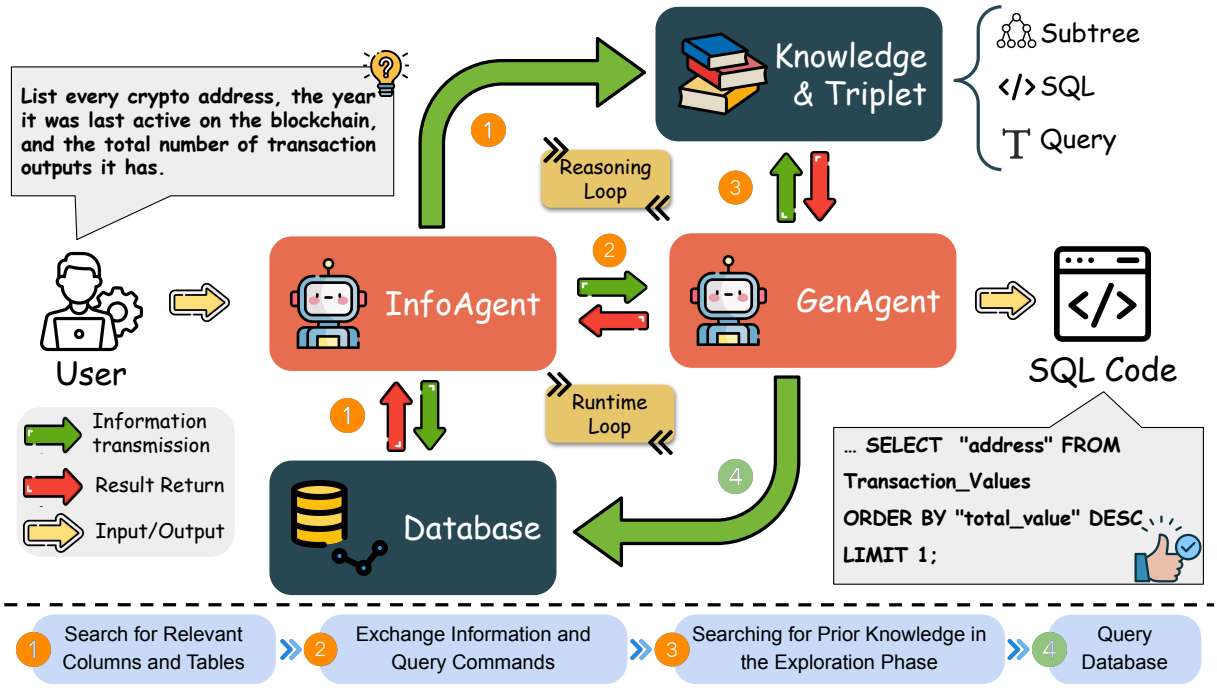


Figure 2: **Deployment Stage: Dual-Agent SQL Synthesis.** The InfoAgent grounds the user query in the database schema via semantic search over pre-computed column embeddings and LLM-driven context expansion. The GenAgent retrieves verified (S, Q, U) triplets from the exploration knowledge base using a joint query-and-schema embedding and synthesizes the final SQL using them as few-shot examples. Their collaborative refinement loop, in which execution feedback simultaneously prunes the schema context and expands overlooked candidates, consistently outperforms a single-agent baseline across all iteration budgets.

to recall additional candidate tables and columns. This dual response ensures that each failed iteration simultaneously tightens the existing context and expands the candidate set in a targeted direction. If execution succeeds, a **Semantic Fidelity Check** verifies that the query result aligns with the user's original intent. This cycle repeats until a valid, semantically aligned query is confirmed or the iteration limit is reached. The formal algorithm is provided in Appendix F.

4 Experiments

To validate the effectiveness and generalizability of SQLAgent, we conduct extensive experiments on two benchmarks: Spider 2.0-Snow and BIRD. We compare against strong baselines, analyze per-component contributions through ablation, and study sensitivity to key hyperparameters and LLM backbones. Full experimental details are provided in Appendix A.

4.1 Experiment Setup

Benchmark. We evaluate on Spider 2.0-Snow (Lei et al., 2025), an enterprise-level benchmark comprising 547 examples across more than 150 databases with approximately 800 columns each. Following the benchmark protocol (Lei et al., 2025), query difficulty is classified by token count into three tiers: Easy (fewer than 80 tokens), Medium (80–159 tokens), and Hard (160 or more tokens).

Evaluation Metrics. We report Execution Accuracy (EX) (Li et al., 2023b; Yu et al., 2018; Lei et al., 2025), which compares the execution results of predicted and ground-truth SQL queries on a given database instance, accounting for the existence of multiple valid queries for a single question. To account for output stochasticity, we also report PASS@K, which measures whether a correct result is produced within K runs. Efficiency is quantified by the average number of LLM calls and database (DB) executions per query.

Baselines. We compare against systems spanning multiple paradigms: Spider-Agent applies multi-agent task decomposition; ReFoRCE combines schema retrieval with iterative self-refinement; AlphaSQL and MARS-SQL are reinforcement-learning-based methods; and CHASE-SQL is a training-free chain-of-thought approach. Our internal ablation baseline implements

a self-refinement mechanism (Deng et al., 2025; Yu et al., 2018): it constructs a textual schema index, retrieves the most relevant DDL, and iteratively refines the SQL using execution feedback until a valid query is produced or a stopping criterion is met.

4.2 Results and Analysis

Comparative Results. We evaluate SQLAgent against all baselines to establish its overall effectiveness. Table 1 presents per-difficulty and efficiency results against Spider-Agent and ReFoRCE. Our full method achieves an overall EX of **25.78%**, surpassing ReFoRCE (20.84%) and Spider-Agent (12.98%). We additionally compare against RL-based methods (AlphaSQL: 7.28%, MARS-SQL: 5.30%) and a training-free reasoning method (CHASE-SQL: 1.28%), confirming that SQLAgent outperforms all baseline categories. The advantage is most pronounced on the Hard subset, where our method achieves **12.21%** while competing methods largely fail. Hard queries require multi-step join reasoning and database-specific operations that are invisible in DDL alone; the knowledge base directly supplies the verified patterns that make such queries tractable. A conceptual comparison with other inference-time frameworks (*e.g.*, Din-SQL, C3, Chess) is provided in Appendix G, and an analysis of failure modes in Appendix H.2.

Ablation Study. To isolate each component's contribution, we conduct a stepwise ablation. The Baseline achieves 14.26% overall EX and scores 0% on Hard queries. Adding the Exploration Stage (w/ Exp. Stage) improves overall accuracy to **20.10%**, confirming the value of the proactively constructed knowledge base. The full method, incorporating the dual-agent Deployment Stage, reaches **25.78%**. In terms of efficiency, our full method averages **5.2** LLM calls and **3.6** DB calls, notably fewer DB interactions than ReFoRCE (3.9) despite higher accuracy, as the structured dual-agent reasoning avoids redundant query executions. A fine-grained component-degradation ablation across five iterative rounds is provided in Appendix I. A notable finding from that study is that removing context pruning causes accuracy to degrade in later refinement rounds rather than stabilize, confirming that accumulated unused schema items introduce noise that can corrupt initially correct queries.

Cost-Amortization Analysis. We verify whether the Exploration Stage's initialization overhead is justified at deployment scale. Table 2 quantifies token costs on a representative large database ($\sim$800 columns). Direct context input fails immediately due to context window overflow. Chunked retrieval incurs no upfront cost but demands 76.2k tokens per query. SQLAgent's one-time exploration cost of 467k tokens is **fully amortized after just 8 queries**, after which its per-query cost of 16.4k tokens is $4.6\times$ cheaper than chunked retrieval. At scale (100 queries), our approach saves over 5.5M tokens while simultaneously achieving a **14.48%** absolute improvement in EX, representing a favorable trade-off for any enterprise deployment scenario. Scalability of the Shared Field Group abstraction is further analyzed in Appendix J.

Dynamic Exploration vs. Static Semantic View. The cost analysis above establishes that the Exploration Stage is efficient at scale; we now verify that its execution-grounded approach is also qualitatively superior to cheaper static alternatives. A natural alternative is to construct a knowledge base by prompting an LLM to synthesize schema descriptions from DDL alone, without any live database interaction. While this incurs no execution cost, it relies entirely on the model's ability to infer schema semantics from structural definitions. We implement this static semantic view baseline using the same GPT-4o backbone to isolate the contribution of execution feedback.

As shown in Table 3, the static baseline achieves only 8.22% overall EX and scores 0.00% on Hard queries. The fundamental limitation is that many database-specific constraints are invisible in DDL and can only be discovered through runtime execution. For instance, the `UNNEST` operation required for nested `VARIANT` fields is not expressed in the schema definition yet is essential for execution. Our dynamic exploration retains only genuinely executable schema combinations by using execution success and failure as objective ground truth, yielding a **17.56%** absolute overall improvement and enabling non-zero performance on Hard queries.

Performance During Iteration. We examine how accuracy evolves across refinement rounds to assess the contribution of the Exploration Stage knowledge base. Figure 3a tracks EX as a function of the iteration budget, comparing the framework with and without the knowledge base. With the knowledge base (red curve), accuracy rises monotonically from 5.5% to a peak of 25.8% at four iterations: each cycle allows the agents to retrieve progressively more relevant examples and refine the

Table 1: **Performance Comparison on Spider 2.0-Snow.** SQLAgent achieves 25.78% overall EX, outperforming all baselines with the most pronounced gain on Hard queries (12.21%). The full method also requires fewer DB calls than ReFoRCE despite higher accuracy. EX (%) is reported per difficulty level; efficiency is measured by average LLM and Database (DB) calls per query.

Method	**Strategy**	**EX (%)**				**Efficiency**	
		Easy	Medium	Hard	Overall	LLM Calls	DB Calls
Spider-Agent	Agentic	24.22	11.38	6.94	12.98	11	3
ReFoRCE	Consensus	49.22	17.00	5.23	20.84	3.5	3.9
SQLAgent	Baseline	32.81	14.57	0.00	14.26	3.0	4.2
	w/ Exp. Stage	41.40	**19.03**	5.81	20.10	4.1	5.2
	Full Method	**57.81**	18.62	**12.21**	**25.78**	5.2	3.6

Table 2: **Cost-Amortization Analysis.** Token costs (k tokens) and Execution Accuracy for a representative large database (∼800 columns). "Initial" is a one-time exploration cost; "Avg./Query" is the per-query inference cost. The exploration overhead is fully amortized after just 8 queries.

Strategy	**Initial**	**Avg./Query**	**Total@10**	**Total@100**	**EX (%)**
Direct Input	—	>Window	Failed		0.00
Chunked Retrieval	0	76.2k	762k	7.62M	11.30
SQLAgent (Ours)	467k	16.4k	**631k**	**2.11M**	**25.78**

schema context. Without the knowledge base (blue curve), accuracy plateaus at 14.1% and sharply degrades after five iterations. Without validated priors, iterative refinement becomes unstable; an accumulation of redundant feedback can lead the agents to revise an initially correct query into an incorrect one. A detailed case study illustrating how the knowledge base resolves complex structures such as nested VARIANT fields and time-sharded relations is provided in Appendix H.

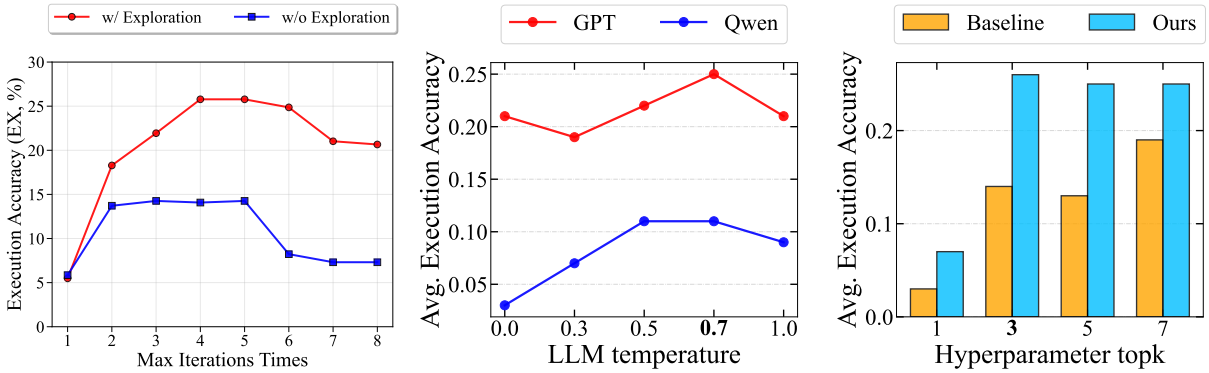
Hyperparameter Analysis. We examine two critical hyperparameters: schema retrieval top-k and LLM temperature. The iteration budget of five rounds is supported by Figure 3a, where performance stabilizes after round four, indicating that additional iterations yield no further benefit. As shown in the right panel of Figure 3b, our method's accuracy rapidly improves and saturates at k=3; beyond this, additional columns yield diminishing returns at higher computational cost. By contrast, the baseline requires a larger k but is constrained by the LLM context window. We therefore set k=3. For temperature (middle panel), a value of 0.7 enables more varied and effective search terms, maximizing EX for the GPT model (Peeperkorn et al., 2024). We set temperature to 0.7 in all experiments.

Generalization on BIRD. To verify that SQLAgent's gains are not specific to Spider 2.0-Snow, we evaluate on the BIRD benchmark (Li et al., 2023c) dev set. Unlike Spider 2.0-Snow, which features heavily sharded enterprise schemas with hundreds of columns per table, BIRD covers a broader variety of real-world databases with more moderate schema complexity and richer natural language variation, making it a complementary test of generalizability. SQLAgent achieves **70.53%** EX, outperforming ReFoRCE at 57.0%, MARS-SQL at 66.8%, and AlphaSQL at 69.7%. This consistent advantage across two structurally distinct benchmarks confirms the generalizability of the two-stage exploration-deployment framework.

LLM Backbone Analysis. To verify that our framework's gains are model-agnostic, we compare the Baseline against the full SQLAgent across four LLM backbones, reporting per-query token cost and EX@8 (OpenAI, 2023; Qwen et al., 2025; Li et al., 2023b). As shown in Table 4, our framework delivers a robust absolute gain of approximately **+10%** across the more powerful models. The improvement is especially pronounced for the smaller Qwen2.5-7B-Instruct model, where our method more than doubles EX@8 from 7.12% to **14.99%**, demonstrating that

Table 3: **Dynamic Exploration vs. Static Semantic View.** Replacing live execution feedback with LLM-only DDL inference completely fails on Hard queries. Dynamic exploration achieves a **17.56%** absolute overall improvement by retaining only genuinely executable schema combinations.

Strategy	Exec. Feedback?	Easy	Medium	Hard	Overall
Static Semantic View	No	20.31	10.07	0.00	8.22
SQLAgent (Ours)	Yes	**57.81**	**18.62**	**12.21**	**25.78**
Improvement		*+37.50*	*+8.55*	*+12.21*	*+17.56*



(a) Performance During Iteration.

(b) Analysis of Two Key Hyperparameters.

Figure 3: **Analysis of the Exploration Stage and Key Hyperparameters.** *Left*: EX as a function of iteration budget. The knowledge base (red) drives accuracy from 5.5% to 25.8% at four iterations; without it (blue), accuracy plateaus at 14.1% and degrades after five iterations as unguided refinement destabilizes initially correct queries. *Middle*: LLM temperature sensitivity: a temperature of 0.7 maximizes EX for the GPT model. *Right*: Schema retrieval top-k analysis: performance saturates at k=3 for our method, while the Baseline requires larger k but is limited by the context window.

Table 4: **Performance Across LLM Backbones.** SQLAgent delivers a consistent ∼+10% absolute EX@8 gain over the baseline across powerful models, and more than doubles performance for Qwen2.5-7B, confirming model-agnostic generalizability. Per-query cost (k tokens) and EX@8 (%) are reported.

LLM	Cost	EX@8 (%)		Δ
		Baseline	Ours	
Qwen2.5-7B	20.8k	7.12	14.99	+7.87
GPT-4o	15.2k	21.57	31.99	+10.42
GPT-5	12.5k	22.49	32.90	+10.70
Claude-Sonnet-4.0	11.6k	29.97	39.12	+9.15

the structured exploration-deployment architecture can effectively compensate for reduced model capacity. An analysis of the knowledge retrieval strategy is provided in Appendix K. Together, these results indicate that schema familiarity has been largely addressed by the Exploration Stage; analysis of remaining failure cases in Appendix H.2 further shows that the principal bottlenecks now lie in multi-step logical reasoning and fine-grained value alignment.

5 Conclusion

We introduce SQLAgent, a framework that emulates a human data engineer's problem-solving process. By first autonomously exploring a database to build context-specific knowledge and then leveraging it for deployment, SQLAgent provides a scalable and adaptive solution for large-scale, enterprise-level databases where conventional methods falter. Extensive experiments validate this approach, demonstrating its effectiveness and robustness across challenging benchmarks. Ultimately, our work represents a significant step towards creating more autonomous and capable agents for data interaction, advancing the goal of democratizing access to complex structured data.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 6250070674) and by the Zhejiang Leading Innovative and Entrepreneur Team Introduction Program (2024R01007).

Limitations

The *Exploration Stage* incurs a one-time initialization cost that may not amortize effectively in cold-start scenarios involving numerous distinct, one-off databases. The framework also requires a live, executable database environment: it is inapplicable in DDL-only settings, and exploratory query execution carries data exposure risks that we recommend mitigating with read-only credentials and organizational access controls. Finally, the *Shared Field Group* abstraction offers limited benefit on highly denormalized schemas, exploration quality is bounded by the backbone LLM's reasoning capacity, and generated SQL may contain logical errors despite the refinement loop, making human oversight advisable in critical deployments.

References

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, and 1 others. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Rémi Coulom. 2007. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83, Berlin, Heidelberg. Springer Berlin Heidelberg.

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, consensus enforcement, and column exploration. *Preprint*, arXiv:2502.00675.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1107–1128, Miami, Florida, USA. Association for Computational Linguistics.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, and 1 others. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

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. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-sql. *The VLDB Journal*, 32(4):905–936.

Hideo Kobayashi, Wuwei Lan, Peng Shi, Shuaichen Chang, Jiang Guo, Henghui Zhu, Zhiguo Wang, and Patrick Ng. 2025. You only read once (YORO): Learning to internalize database knowledge for text-to-SQL. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1889–1901, Albuquerque, New Mexico. Association for Computational Linguistics.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg. Springer Berlin Heidelberg.

LangChain AI. 2024. *LangGraph: Building stateful, multi-actor applications with LLMs*.

langgenius. 2023. *Dify: An open-source LLM app development platform*.

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. *Preprint*, arXiv:2411.07763.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang

Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023b. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023c. 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*, 36:42330–42357.

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.

Mohammadhossein Malekpour, Nour Shaheen, Foutse Khomh, and Amine Mhedhbi. 2024. Towards optimizing sql generation via llm routing. *Preprint*, arXiv:2411.04319.

Meta Fundamental AI Research (FAIR) Diplomacy Team, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, and 1 others. 2022. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074.

Suvir Mirchandani, Fei Xia, Pete Florence, Brian Ichter, Danny Driess, Montserrat Gonzalez Arenas, Kanishka Rao, Dorsa Sadigh, and Andy Zeng. 2023. Large language models as general pattern machines. *arXiv preprint arXiv:2307.04721*.

Neo4j, Inc. 2024. *Neo4j Graph Database*. Accessed: 2025-12-20.

OpenAI. 2023. GPT-4 technical report. Technical report, OpenAI.

OpenAI. 2024a. Hello gpt-4o. Accessed: 2025-12-20.

OpenAI. 2024b. New embedding models and api updates. Accessed: 2024-05-20.

Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. 2024. Is temperature the creativity parameter of large language models? *Preprint*, arXiv:2405.00492.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, Seattle, United States. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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 Comput. Surv.*, 58(2).

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

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

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA. Association for Computing Machinery.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023a. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023b. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language

models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. *Preprint*, arXiv:2407.16741.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang (Eric) Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. Technical Report MSR-TR-2023-33, Microsoft.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Semih Yavuz, Izzeddin Gur, Yu Su, and Xifeng Yan. 2018. What it takes to achieve 100% condition accuracy on WikiSQL. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1702–1711, Brussels, Belgium. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3501–3532.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2022a. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*.

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022b. Active example selection for in-context learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9134–9148, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

A Experimental Details

Throughout our experiments, we retrieve the top-3 most relevant database tables ($k = 3$) to populate the context and set the maximum number of self-refinement attempts to 5. To ensure a fair comparison, these hyperparameter settings are kept consistent for both the baseline and our proposed method.

Unless otherwise specified, all experiments utilize GPT-4o as the backbone LLM (OpenAI, 2024a). The temperature for all LLMs was set to 0.7. For experiments involving local model inference (e.g., open-source baselines), we utilized a server equipped with two NVIDIA RTX 4090 GPUs.

The multi-agent framework is built upon the LangGraph library (LangChain AI, 2024). Vector embeddings are generated using the `text-embedding3small` model from OpenAI (OpenAI, 2024b), and all vectorization tasks are subsequently handled by the Faiss library (Johnson et al., 2019). Our data modeling framework is built entirely on Neo4j (Neo4j, Inc., 2024), and we employ the Cypher query language for all graph traversal.

Artifact Licenses and Terms. We utilize two public benchmarks for evaluation. The Spider 2.0 benchmark (Lei et al., 2025) is distributed under the **CC BY-SA 4.0** license. The BIRD benchmark (Li et al., 2023c) is distributed under the **CC BY-SA 4.0** license. We confirm that our use of both datasets is consistent with their intended purpose for research. The code for SQLAgent, along with the constructed knowledge base, is released under the **MIT License** and is available in the supplementary material.

Evaluation Protocol. Due to the significant computational cost of running the full benchmark, all reported results reflect a single evaluation run. The deterministic nature of SQL execution over a fixed dataset ensures minimal variance across runs.

B Implementation Details of Deployment Stage

B.1 Schema and Query Vectorization

As mentioned in Section 3.3, we employ semantic retrieval to bridge the gap between natural language and database schemas.

Schema Embedding. To enable efficient retrieval, each column c in the database is serialized into a descriptive string following the template: `"[Name]: name; [Type]: type; [Desc]: comment"`.

Knowledge Base Embedding. For the Knowledge Base constructed in the Exploration Stage, we vectorize the SQL component (Q) of each triplet (S, Q, U). We utilize a code-specific embedding model (e.g., `textembedding3small` or `unixcoder`) to capture the structural and semantic features of the SQL queries. This allows the GenAgent to retrieve examples based on structural similarity rather than just keyword overlap.

B.2 Agent Prompt Design

The performance of SQLAgent relies on specific prompt instructions designed for the InfoAgent and GenAgent.

InfoAgent Context Expansion. In the deployment stage, the InfoAgent performs a second expansion step to infer implicit dependencies. The LLM is prompted to act as a database expert. For example, if the initial retrieval contains `users.user_name` and `orders.order_amount`, the prompt instructs the model to identify the necessary join keys (e.g., `users.user_id` and `orders.customer_id`) to link these tables. A simplified version of the instruction is:

> *"Given the user query and the retrieved schema fragments, analyze the relationships. If two tables are required but no join condition is present, identify and add the primary/foreign keys necessary to form a valid SQL join."*

GenAgent Synthesis. The GenAgent receives a composite prompt containing:

1. **Role Definition:** "You are an expert SQL data analyst."
2. **Schema Context:** The refined JSON structure from the InfoAgent.
3. **Few-Shot Examples:** The top-k retrieved triplets (S, Q, U) from the Knowledge Base.
4. **User Query:** The target natural language question.

This structured prompt ensures the model adheres to the specific syntax and schema constraints of the target database.

C Definition of Database Structure

As introduced in the main methodology, we represent the relational database as a traversable graph structure to facilitate autonomous exploration. This model is composed of distinct node and relationship types that capture the hierarchical and relational nature of the database schema. This appendix provides a formal definition of each component, their respective properties, and an overview of the graph's statistical composition.

C.1 Graph Schema Overview

The graph model consists of five distinct node types and five relationship types that define their connections. The primary node types are:

- **Database**: The root node representing the entire database instance.
- **Schema**: Represents a schema or dataset within the database.
- **Table**: Represents a single table.
- **Field**: Represents a column within a table.
- **Shared Field Group**: Our proposed abstraction for a reusable set of fields shared by multiple tables.

The relationships between these nodes define the structural integrity of the graph. Table 5 outlines the valid connections between node types.

C.2 Node Property Definitions

Each node in the graph contains a set of properties that store its metadata. The following tables detail the properties for each of the five node types.

C.3 Graph Composition Statistics

To provide a sense of scale, Table 11 summarizes the composition of the graph constructed from our experimental datasets. The statistics highlight the prevalence of fields, which constitute over 90% of all nodes.

Table 5: Relationship types defining the connections between nodes in the graph schema.

Start Node Type	Relationship Type	End Node Type
Database	HAS_SCHEMA	Schema
Schema	HAS_TABLE	Table
Table	USES_FIELD_GROUP	SharedFieldGroup
Table	HAS_UNIQUE_FIELD	Field
SharedFieldGroup	HAS_FIELD	Field

Table 6: Properties of the Database node.

Property	Description	Example
<id>	Internal unique identifier for the node.	15313
name	The name of the database instance.	OPEN_TARGETS_PLATFORM_2
type	Node type specifier.	database

Table 7: Properties of the Schema node.

Property	Description	Example
<id>	Internal unique identifier for the node.	5430
database	Name of the parent database.	NOAA_DATA
description	A description of the schema, if available.	-
name	The name of the schema.	NOAA_SIGNIFICANT_EARTHQUAKES
type	Node type specifier.	schema

Table 8: Properties of the Table node.

Property	Description	Example
<id>	Internal unique identifier for the node.	185
database	Name of the parent database.	COVID19_USA
ddl_summary	A summary of the table's DDL statement.	Table with 245 columns
fullname	The fully qualified name of the table.	CENSUS_BUREAU_ACS_2...
name	The name of the table.	SCHOOLDISTRICTSECONDARY...
schema	Name of the parent schema.	CENSUS_BUREAU_ACS
type	Node type specifier.	table

Table 9: Properties of the SharedFieldGroup node.

Property	Description	Example
<id>	Internal unique identifier for the node.	2079
database	Name of the parent database.	NEW_YORK_PLUS
description	Auto-generated description of the group.	FieldGroup_3ebfe5f9...
field_count	Number of fields encapsulated in this group.	24
field_hash	An MD5 hash of the sorted field names, used to identify unique groups.	3ebfe5f9...
name	Auto-generated name for the group.	FieldGroup_3ebfe5f9
schema	Name of the parent schema.	NEW_YORK_TAXI_TRIPS
type	Node type specifier.	shared_field_group

Table 10: Properties of the Field node.

Property	Description	Example
<id>	Internal unique identifier for the node.	18
database	Name of the parent database.	NEW_YORK
description	A description of the field, if available.	-
name	The name of the column/field.	contributing_factor_vehicle_4
node_type	Specifies if the field is part of a group or unique to a table.	unique_field
sample_data	Sample data from this column.	-
schema	Name of the parent schema.	NEW_YORK
table	Name of the parent table (for unique fields).	NYPD_MV_COLLISIONS
type	The data type of the field.	TEXT

Table 11: Statistical overview of the constructed graph representation.

Component	Count	Percentage
Node Types		
Database	151	0.2%
Schema	267	0.3%
Table	7,848	8.7%
SharedFieldGroup	542	0.6%
Field	81,298	90.2%
Total Nodes	**90,106**	**100.0%**
Relationship Types		
HAS_SCHEMA	534	-
HAS_TABLE	15,696	-
USES_FIELD_GROUP	11,478	-
HAS_UNIQUE_FIELD	104,808	-
HAS_FIELD	57,788	-
Total Relationships	**190,304**	-

A key component of our model is the SharedFieldGroup, designed to reduce redundancy. Our analysis confirms its effectiveness:

- **Total Groups:** There are **542** unique SharedFieldGroup nodes.
- **Average Fan-out:** On average, each group is linked to **10.6** tables, indicating frequent reuse.
- **Maximum Fan-out:** The most utilized group is shared by **334** distinct tables.

This analysis demonstrates the high prevalence of redundant schema structures in large-scale databases and validates our abstraction's utility in creating a more compact and efficient representation for exploration.

D Identifying Shared Field Groups

The identification of Shared Field Group nodes is a critical preprocessing step designed to abstract and consolidate recurring schema structures within the database. This process ensures that tables with identical field compositions are linked to a single, canonical group node, thereby reducing redundancy in the graph representation. The methodology is executed in two primary phases: (1) generating a unique signature for each distinct set of fields, and (2) applying a greedy algorithm to select a final, non-overlapping set of shared groups.

Phase 1: Field Group Signature Generation To consistently identify tables that share an identical set of fields, we first compute a content-based signature for the field structure of each table. This signature is derived from the names and data types of all columns within a table.

The process, detailed in Algorithm 1, involves creating a canonical string representation for the set of fields. Each field is formatted as a string concatenation of its name and type (*e.g.*, "user_id:INTEGER"). To ensure that the signature is independent of the original column order, these formatted strings are sorted alphabetically before being joined into a single, delimited string. Finally, the MD5 hash function is applied to this canonical string to produce a compact and unique 128-bit signature. Any two tables that yield the same signature are considered to have structurally identical schemas.

Phase 2: Greedy Selection of Non-Overlapping Groups After an initial pass over all tables, we obtain a collection of potential shared groups, each identified by its unique signature and associated with a list of tables that match it. A subsequent optimization phase is necessary to produce a final, non-overlapping set of SharedFieldGroups. This is crucial because complex schemas may contain

Algorithm 1 Field Group Signature Generation

Input: A table's field set $F_{table} = \{(n_1, t_1), (n_2, t_2), \ldots, (n_m, t_m)\}$, where n_i is a field name, t_i is its data type, and m is the number of fields.
Output: A unique signature string S_{group}.

function GENERATESIGNATURE(F_{table})
  Initialize an empty list L_{fields}
  for each field (n_i, t_i) in F_{table} **do**
    $s_i \leftarrow$ Concatenate(n_i, " : ", t_i) ▷ Format as "name:type"
    Append s_i to L_{fields}
  end for
  Sort L_{fields} alphabetically
  $S_{canonical} \leftarrow$ Join(L_{fields}, "|") ▷ Create a canonical, delimited string
  $S_{group} \leftarrow$ MD5($S_{canonical}$) ▷ Compute the MD5 hash
  return S_{group}
end function

nested or overlapping field structures.

We employ a greedy selection algorithm, detailed in Algorithm 2, to resolve this. The core principle is to prioritize the most significant and impactful shared structures. First, all potential groups are sorted in descending order based on two criteria: primarily by the number of tables they encompass, and secondarily by the number of fields they contain (as a tie-breaker). This heuristic prioritizes groups that represent the most widespread schema patterns.

The algorithm then iterates through this sorted list. For each candidate group, it checks if any of its member tables have already been assigned to a previously selected group. If there is no overlap, the group is added to the final set, and all of its member tables are marked as assigned. This process ensures that each table can belong to at most one `SharedFieldGroup`, resulting in an unambiguous and efficient final graph structure.

E Detailed Action Space

This section provides a detailed description of the discrete action space used by the LLM in the exploration stage. At each step of the tree search, the LLM selects one of the following actions to incrementally construct a SQL query based on the current query state and schema context.

- **Select Unused Column:** Adds a new, previously unselected column to the query's 'SELECT' statement. This action progressively expands the breadth of information retrieved by the query.
- **Add Predicate Constraint:** Applies a filter condition to an existing column, typically by adding a 'WHERE' clause. This action narrows the scope of the query, allowing it to focus on specific subsets of data that meet certain criteria.
- **Introduce Join:** Connects the current set of tables to a new table based on foreign key relationships or 'Shared Field Group' linkages. This action is fundamental for exploring relationships across different tables and synthesizing information from multiple sources.
- **Apply Aggregation Function:** Applies a summary function (*e.g.*, 'COUNT', 'SUM', 'AVG', 'MAX', 'MIN') to a previously selected column. This action transforms the query's purpose from simple record retrieval to data summarization, enabling the model to

Algorithm 2 Greedy Selection of Shared Field Groups

Input: A collection of all potential groups $\mathcal{P}_{all}$, where each group $g \in \mathcal{P}_{all}$ has a signature, a set of fields, and a set of matching tables T_g.
Output: A final, non-overlapping set of shared groups $\mathcal{P}_{final}$.

function SELECTGROUPS($\mathcal{P}_{all}$)
  $\mathcal{P}_{filtered} \leftarrow \{g \in \mathcal{P}_{all} \mid |T_g| \geq 2\}$ ▷ Consider only groups with at least two tables
  Sort $\mathcal{P}_{filtered}$ in descending order by $|T_g|$, then by number of fields.
  Initialize $\mathcal{P}_{final} \leftarrow \emptyset$
  Initialize set of assigned tables $T_{assigned} \leftarrow \emptyset$
  for each group g in sorted $\mathcal{P}_{filtered}$ **do**
    $T_{current} \leftarrow$ the set of tables in group g.
    if $T_{current} \cap T_{assigned} = \emptyset$ **then** ▷ Check for overlap with already assigned tables
      $\mathcal{P}_{final} \leftarrow \mathcal{P}_{final} \cup \{g\}$ ▷ Add group to the final set
      $T_{assigned} \leftarrow T_{assigned} \cup T_{current}$ ▷ Mark tables as assigned
    end if
  end for
  return $\mathcal{P}_{final}$
end function

understand the scale and distribution of the data.

- **Add Group By Clause:** Groups the result set by one or more selected non-aggregated columns. This action is typically used in conjunction with aggregation functions to perform categorical analysis, such as calculating metrics for different segments of the data (*e.g.*, "total sales per region").
- **Add Ordering Clause:** Sorts the final result set based on a specified column, either in ascending ('ASC') or descending ('DESC') order via an 'ORDER BY' clause. This enables the discovery of extremes, such as top-performing items or most recent events.
- **Add Having Clause:** Applies a filter condition to the results of a 'GROUP BY' aggregation. Unlike a 'WHERE' clause which filters rows before aggregation, 'HAVING' filters entire groups after aggregation, enabling more complex analytical questions like identifying categories that meet a certain threshold (*e.g.*, "customers with more than 5 orders").

F Algorithm for Dual-Agent SQL Synthesis

This section provides a detailed algorithmic implementation of the dual-agent framework for SQL synthesis, as described in the Deployment Stage of our methodology. The process is designed as an iterative loop where the **InfoAgent** and **GenAgent** collaborate to refine the context and generate the final SQL query. The InfoAgent is responsible for schema grounding and context management, while the GenAgent leverages the acquired knowledge base to synthesize the query. Algorithm 3 formalizes this collaborative workflow.

Algorithmic Details The algorithm details the iterative process of context refinement and query generation managed by the dual-agent framework. The process begins with an **initialization** phase (Lines 1-4), where the iteration counter, a success flag, and an empty structure for feedback information are prepared. The `feedback_info` variable is crucial for passing insights from a failed attempt to the next iteration.

Each cycle within the main loop starts with the **InfoAgent** performing schema grounding and expansion (Lines 6-9). It first uses an LLM to extract semantic keywords from the user's utterance and performs a semantic search to retrieve an initial set of schema components. Since this set may be incomplete, a second LLM-driven step expands it by reasoning about logical necessities, such as join keys. Following this, a crucial **context pruning** step occurs (Line 9), where the InfoAgent uses feedback from any previous failed iteration to remove unused schema components, thus narrowing the search space.

The refined context is then passed to the **GenAgent**, which conducts **knowledge retrieval** (Lines 11-12) by using the user's intent to find the most relevant triplets from the knowledge base $\mathcal{K}$. With these triplets as in-context examples, the GenAgent proceeds to **SQL synthesis** (Line 13), generating a candidate query. This query then undergoes **execution and validation** (Lines 15-24). This step has three possible outcomes. An **Execution Failure** (*e.g.*, a syntax error) or a **Semantic Mismatch** (where the query output does not match the user's intent) results in failure feedback being stored in `feedback_info` for the next loop. A **Success** occurs if the query executes correctly and passes the fidelity check, which sets the success flag and terminates the loop. The entire process continues until a valid query is found or the maximum number of iterations is reached, as defined by the **termination** condition (Lines 27-31), after which the final query or a failure signal is returned.

G Extended Comparison with Inference-time Frameworks

To further clarify the positioning of SQLAGENT relative to recent state-of-the-art (SOTA) Text-to-SQL frameworks, we provide a detailed conceptual comparison in this section.

G.1 Proactive Knowledge Construction vs. Inference-time Prompting

Frameworks such as **Din-SQL**, **C3**, and **Chess** primarily focus on optimizing the *inference-time* reasoning process. They assume that given a sufficiently sophisticated prompt or decomposition strategy, a frozen LLM can correctly map natural language to an unseen database schema.

In contrast, SQLAGENT operates on the premise that for complex enterprise-level databases, the primary bottleneck is the agent's lack of familiarity with unique schema semantics and intricate join paths. Our framework decouples knowledge acqui-

Algorithm 3 Dual-Agent SQL Synthesis Workflow

Input: User Utterance U_{in}, Database Schema Graph $\mathcal{G}$, Knowledge Base $\mathcal{K}$, Database $\mathcal{D}$, Max Iterations N_{max}
Output: Final SQL Query Q_{final} or `failure`

function DUALAGENTSYNTHESIS($U_{in}, \mathcal{G}, \mathcal{K}, \mathcal{D}, N_{max}$)
  $i \leftarrow 0$
  $is\_success \leftarrow$ **false**
  $feedback\_info \leftarrow \emptyset$ ▷ Stores context from failed attempts for pruning
  while $i < N_{max}$ **and not** $is\_success$ **do**
    ▷ **InfoAgent**: Schema Grounding & Expansion
    $keywords \leftarrow$ LLM.ExtractKeywords(U_{in})
    $C_{initial} \leftarrow$ SemanticSearch($\mathcal{G}, keywords$) ▷ Retrieve initial top-k schema components
    $C_{expanded} \leftarrow$ LLM.ExpandContext($U_{in}, C_{initial}$) ▷ Reason and add necessary related components
    $C_{ctx} \leftarrow C_{expanded} \setminus$ PruneUnused($feedback\_info$) ▷ Refine context based on last failure
    ▷ **GenAgent**: Knowledge Retrieval & SQL Synthesis
    $Q_{embed} \leftarrow$ EmbedQueryIntent(U_{in}, C_{ctx})
    $\mathcal{K}_{retrieved} \leftarrow$ SimilaritySearch($\mathcal{K}, Q_{embed}$) ▷ Retrieve top-k triplets
    $Q_{cand} \leftarrow$ LLM.GenerateSQL($U_{in}, C_{ctx}, \mathcal{K}_{retrieved}$) ▷ Generate candidate query
    ▷ Execution and Validation Feedback Loop
    $result, error \leftarrow$ ExecuteSQL($\mathcal{D}, Q_{cand}$)
    if $error \neq \emptyset$ **or** $result$ is empty **then**
      $feedback\_info \leftarrow (Q_{cand}, C_{ctx},$ "Execution Failed") ▷ Package feedback for context pruning
    else
      $is\_aligned \leftarrow$ LLM.CheckFidelity($U_{in}, result$) ▷ Semantic fidelity check
      if $is\_aligned$ **then**
        $Q_{final} \leftarrow Q_{cand}$
        $is\_success \leftarrow$ **true**
      else
        $feedback\_info \leftarrow (Q_{cand}, C_{ctx},$ "Semantic Mismatch")
      end if
    end if
    $i \leftarrow i + 1$
  end while
  if $is\_success$ **then**
    return Q_{final}
  else
    return `failure`
  end if
end function

sition from query generation:

- **Offline Exploration Stage**: SQLAGENT autonomously interacts with the live database to construct an executable knowledge base of triplets (S, Q, U). This stage acts as an "automated data engineer" learning the database's specific logic before any user query is received.
- **Deployment Stage**: The agent retrieves these validated, database-specific examples as in-context prompts. This effectively addresses the "cold-start" problem where inference-only methods often struggle due to ambiguous column names or undocumented relationships.

G.2 Amortized Cost Analysis

While the Exploration Stage introduces an initialization overhead, this cost is **amortized** over the lifespan of the database application. In enterprise settings, database schemas are relatively stable, whereas user queries are frequent and repetitive. By investing in proactive exploration once, SQLAGENT provides a more robust and accurate interface for all subsequent interactions, making the exploration cost a favorable trade-off for persistent data platforms.

In this section, we provide a qualitative analysis to demonstrate how the database-specific knowledge acquired in the Exploration Stage facilitates complex SQL synthesis, followed by a detailed breakdown of the current failure modes observed in our experiments.

H Case Study & Error Analysis

In this section, we provide a qualitative analysis to demonstrate how the database-specific knowledge acquired in the Exploration Stage facilitates complex SQL synthesis, followed by a detailed textual breakdown of the current failure modes observed in our experiments.

H.1 Case Study: Resolving Nested Structures and Sharded Relations

The primary advantage of SQLAGENT is its ability to proactively discover intricate join paths and undocumented schema semantics that are often invisible to zero-shot models. We illustrate this with a representative query from the `GA4_OBFUSCATED_SAMPLE_ECOMMERCE` dataset, which is characterized by time-sharded tables and nested `VARIANT` fields.

User Question: "I want to know the preferences of customers who purchased the 'Google Navy Speckled Tee' in December 2020. What other product was purchased with the highest total quantity alongside this item?"

The Challenge: The database stores daily events in separate sharded tables (e.g., `EVENTS_202012*`). SQLAGENT utilizes **Shared Field Groups** to abstract these into a single traversable template, significantly reducing search complexity. Furthermore, product information is encapsulated within a `VARIANT` column named `ITEMS`. Accessing the `item_name` requires an `UNNEST` operation, a requirement that is not explicitly stated in the schema's DDL but is essential for execution.

Retrieved Exploration Triplet: During the Exploration Stage, the agent validated the following triplet:

- **Schema Sub-structure** (S): `{events_*, items (VARIANT), UNNEST(items) AS i}`
- **SQL Fragment** (Q): `SELECT user_pseudo_id, i.item_name FROM ga4.events_*, UNNEST(items) AS i WHERE i.item_name IS NOT NULL`
- **Description** (U): "How to extract individual product names from the nested items array in GA4 tables."

Success vs. Failure: Without this triplet, the baseline model attempted to access `items.item_name` directly, resulting in a syntax error. By leveraging the triplet, the **GenAgent** correctly implemented the `UNNEST` pattern within a Common Table Expression (CTE), leading to a successful execution.

H.2 Detailed Error Analysis

Manual analysis of 100 failed samples in the Hard category (74.22% unresolved) reveals that remaining bottlenecks primarily stem from the inherent complexity of enterprise-scale Text-to-SQL rather than schema unfamiliarity. These challenges are twofold: (i) **Reasoning Constraints**, where extreme logical nesting (*e.g.*, $\geq$ 4 subqueries) or excessive statement length ($>$ 100 lines) occasionally exceed the backbone LLM's multi-step reasoning capacity and context window; and (ii) **Fine-grained Semantic Grounding**, involving precise value-to-cell alignment (*e.g.*, date formatting) and disambiguation between functionally distinct but semantically similar fields in dense schemas exceeding 800 columns. These findings suggest that while the Exploration Stage effectively bridges the "schema familiarity" gap, the next frontier for autonomous data agents lies in advancing multi-step logical synthesis and precise value alignment within massive-scale enterprise environments.

I Detailed Component-Degradation Ablation

To precisely quantify the contribution of each architectural component and the dynamics of the iterative refinement loop, we conduct a component-degradation ablation study tracking Execution Accuracy (EX, %) across five iterative rounds. Table 12 presents seven configurations obtained by

systematically removing or replacing individual modules of the full SQLAgent framework.

The results reveal four key insights:

Necessity of Exploration Knowledge. Removing the knowledge base ("Remove GenAgent Knowledge") caps peak accuracy at 14.26%, confirming that raw DDL alone is insufficient for complex multi-step queries.

Power of the Refinement Loop. The full method surges from 5.48% (Round 1) to a stable 25.78% (Round 4–5), demonstrating that iterative context refinement is essential for resolving complex queries that cannot be answered in a single pass.

InfoAgent Component Dynamics. *Context Expansion* is critical for discovering implicit join paths: removing it reduces peak accuracy by 2.93 points. More notably, removing *Context Pruning* causes accuracy to actively degrade in later rounds (from 21.02% at R3 to 16.45% at R5), because accumulating unused schema items overwhelms the context window and introduces noise, causing agents to revise initially correct queries into incorrect ones.

GenAgent Component Dynamics. Structural vector retrieval outperforms keyword matching (+5.49% at peak), confirming that semantic similarity over SQL structure is a more effective retrieval signal than lexical overlap. The *Semantic Fidelity Check* acts as a guardrail against false positives, specifically queries that execute successfully but answer a different question than the user intended, contributing 4.76% to peak accuracy.

J Shared Field Group: Scalability Analysis

J.1 Exploration Termination Criteria

The exploration stage terminates based on two criteria, regardless of database size: (1) **Field coverage**: exploration halts when 90% of all fields have been incorporated into at least one validated triplet; (2) **Iteration budget**: a hard cap of 300 total attempts prevents unbounded exploration on extremely large schemas. These criteria balance thoroughness against computational cost and can be tuned based on deployment requirements.

J.2 Effect of Removing Shared Field Group

The Shared Field Group abstraction is fundamental to making exploration tractable on large-scale enterprise databases. To quantify its impact, we conduct ablation experiments at two levels of granularity.

Single-database analysis (BLS_QCEW). Table 13 reports results on the `BLS_QCEW` database, a highly sharded schema representative of enterprise time-series data stores.

Without the abstraction, 22,892 redundant field nodes must be individually traversed; the agent exhausts its context window on repetitive, semantically identical paths rather than exploring diverse query structures. The Shared Field Group compresses these into a single canonical node, achieving a **98.6% reduction** in graph size and a **45.0-point** improvement in exploration success rate.

Full benchmark analysis. Table 14 reports statistics across all 151 databases in our benchmark.

At scale, the abstraction reduces total graph nodes from 1,030,327 to 90,106 (an $11.4\times$ compression) and more than doubles the average exploration success rate. Beyond efficiency, this redundancy reduction directly improves *diversity*: without shared groups, the agent wastes its exploration budget generating semantically identical queries for every sharded table (e.g., `sales_day1`, `sales_day2`), crowding out coverage of structurally distinct schema regions.

K Knowledge Retrieval Strategy Analysis

An alternative to retrieving structured (S, Q, U) triplets from the knowledge base is to directly vectorize and retrieve the raw exploration trajectories, including all intermediate trial-and-error steps, as context for the InfoAgent. Table 15 compares these two retrieval strategies.

Direct vectorization of raw trajectories underperforms by **6.96 points**. Raw exploration logs contain significant noise: failed attempts, partial queries, and repetitive backtracking steps that do not represent valid schema knowledge. Including this noise degrades retrieval precision and introduces misleading in-context examples. By contrast, our distillation process, which retains only triplets satisfying syntactic validity, execution success, and non-empty result criteria, produces a compact, high-signal knowledge base that enables precise few-shot retrieval during deployment.

Table 12: **In-depth Component-Degradation Ablation across Iteration Rounds.** EX (%) is reported at each round of the dual-agent refinement loop. Configurations are obtained by removing or replacing individual modules from the full method.

Configuration	Change Description	R1	R2	R3	R4	R5
Full Method (SQLAgent)	Complete dual-agent framework	5.48	18.28	21.94	**25.78**	**25.78**
Replace InfoAgent	Replace with simple semantic retrieval	5.48	11.88	16.45	18.65	19.20
w/o Context Expansion	Skip LLM-based dependency inference (e.g., foreign keys)	5.48	15.54	20.11	22.49	22.85
w/o Context Pruning	Do not remove unused schema items during refinement	5.48	17.37	21.02	19.20	16.45
Remove GenAgent Knowledge	Remove exploration knowledge; input raw DDL only	5.85	13.71	14.26	14.08	14.26
Replace Semantic Retrieval	Use keyword matching instead of vector embeddings	5.12	12.80	17.37	19.74	20.29
w/o Semantic Fidelity Check	Accept any successfully executed SQL without intent check	5.48	14.63	19.20	21.02	21.02

Table 13: **Shared Field Group Impact on BLS_QCEW Database.** The abstraction reduces graph nodes by 98.6% and more than doubles the exploration success rate.

Configuration	Graph Nodes	Search Space	Success Rate
Without Shared Group	23,010 (118T+22,892F)	Intractable	33.5%
With Shared Group (Ours)	**313** (118T+1G+194F)	Converged	**78.5%**

Table 14: **Shared Field Group Impact Across Full Benchmark.** Removing the abstraction inflates the total graph by $11.4\times$ and more than halves the exploration success rate.

Configuration	Total Nodes	Avg. Success Rate
Without Shared Group	1,030,327	36.17%
With Shared Group (Ours)	**90,106**	**74.69%**

Table 15: **Retrieval Strategy Comparison.** Structured triplet retrieval substantially outperforms direct vectorization of raw exploration trajectories.

Strategy	Characteristics	EX (%)
Direct Vectorization	Raw exploration trajectories (noisy, redundant)	18.82
Ours ((S, Q, U) Triplets)	Structured, validated, purified experience	**25.78**

L LLM Usage

This statement is to disclose the use of large language models (LLMs) in preparing this manuscript, in accordance with ACL policy. The use of LLMs was strictly limited to improving the language and readability of the text, including grammar correction and stylistic polishing. LLMs did not contribute to any core scientific aspects of this work, such as research ideation, experimental design, or data analysis. All intellectual contributions are the original work of the authors, who assume full responsibility for the final content.