DecoSearch: Complexity-Aware Routing and Plan-Level Repair for Text-to-SQL

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in translating natural language to SQL, yet existing methods still falter on complex queries requiring multi-step, data-aware reasoning. We introduce DecoSearch, a training-free framework that addresses this by routing each query to the appropriate level of reasoning effort. A lightweight Schema Selector first prunes the full database schema to the relevant tables and columns. An LLM Judger then decides whether the question requires decomposition: straightforward questions follow a direct generation path and complex ones are escalated to a Directed Acyclic Graph (DAG) of atomic subquestions, each solved by a targeted SQL generation step. A RAG component grounds the decomposer with semantically similar training examples, and a Topology Refiner restructures the reasoning plan when execution failures signal a flawed decomposition rather than a fixable SQL error. DecoSearch achieves 70.53% execution accuracy on BIRD and 88.31% on Spider with a DeepSeek backbone, surpassing all training-free baselines while consuming an order of magnitude fewer tokens than competing methods. It also functions as a model-agnostic wrapper, consistently improving fine-tuned SQL generation backbones without any modification to the pipeline.

1 Introduction

The conversion of natural language questions into executable SQL (Text-to-SQL) is a fundamental challenge at the intersection of natural language understanding and database systems (Katsogiannis-Meimarakis and Koutrika, 2023; Luo et al., 2025; Yu et al., 2018; Li et al., 2023). The advent of Large Language Models (LLMs) has marked a new era of capability in this domain, enabling zero-shot or few-shot translation with impressive fluency (Rajkumar et al., 2022; Gao et al., 2024; Pourreza and Rafiei, 2023; Talaei et al., 2024). However, the performance of these models degrades significantly when confronted with complex questions, those requiring nested queries, multi-step logical inference, or an awareness of the database's underlying content and structure (Li et al., 2023, 2024), or large databases with multiple tables that requires complex SQL queries with multiple JOIN operations (Yu et al., 2018; Li et al., 2024). For these problems, a single-pass, autoregressive generation process is often insufficient (Scholak et al., 2021; Pourreza and Rafiei, 2023; Talaei et al., 2024; Pourreza et al., 2024), as it fails to explore the vast combinatorial space of possible SQL queries and cannot recover from early generation errors.

Recent work has attacked this brittleness from several angles. Decomposition-based methods break a complex question into simpler subproblems, improving the tractability of each step (Pourreza and Rafiei, 2023; Zhou et al., 2023; Eyal et al., 2023; Wang et al., 2025a; Xie et al., 2024). Self-refinement approaches use execution feedback to iteratively debug generated SQL (Madaan et al., 2023; Gou et al., 2024; Chen et al., 2024; Askari et al., 2025; Zhai et al., 2025). Multi-candidate frameworks generate diverse SQL candidates and select the best one (Wang et al., 2023; Pourreza et al., 2024; Sun et al., 2023; Lee et al., 2025; Borchmann and Wydmuch, 2025). Complexity-aware routing strategies allocate different pipelines to queries of different difficulty (Zhu et al., 2025; Guo et al., 2025). Each of these strategies addresses a real failure mode, but each operates at a single level of the problem. A self-debugging system can fix a syntactic error in a SQL query, but it cannot recognize that the underlying reasoning plan was flawed. A decomposition system can produce a sensible plan, but if one subproblem turns out to be unsolvable as posed, it has no mechanism to revise its own plan. A multi-candidate system offers breadth, but it does not

adaptively concentrate its computational budget on the most promising direction.

We observe that failures in complex Text-to-SQL generation are not monolithic. They arise from qualitatively different root causes that demand different interventions. Some failures are *implementation-level*: the SQL contains a bug such as a wrong join condition or an incorrect filter, and an LLM debugger equipped with execution feedback can repair it directly. Other failures are *strategy-level*: the chosen SQL formulation style (e.g., nested subqueries versus CTEs) is a poor fit for the problem, and a structurally different approach would succeed where iterative debugging of the current one would not. Still other failures are *plan-level*: the high-level reasoning plan itself is flawed, a dependency is missing, or the decomposition is too coarse, and no amount of SQL-level repair can compensate. Existing methods typically address at most one of these failure types. No current Text-to-SQL framework systematically diagnoses the level at which a failure occurs and routes it to the appropriate repair mechanism.

We introduce DecoSearch, a training-free framework that organizes Text-to-SQL generation as a multi-stage adaptive workflow with hierarchical failure routing. The framework employs a Judger-led escalation strategy: for every question, a specialized LLM judger evaluates whether the query can be solved directly or requires structural decomposition. For manageable questions, the system follows a Direct Path that generates and executes a single SQL query, keeping latency and cost low. For queries judged as complex, DecoSearch escalates to the plan level, decomposing the question into a Directed Acyclic Graph (DAG). Each sub-question is solved independently, and persistent execution failures at the query level trigger automatic topology refinement to restructure the reasoning plan. This ensures computational resources are spent proportionally to the depth of the failure.

Our primary contributions are as follows:

- We identify that failures in complex Text-to-SQL arise at distinct levels of abstraction (implementation, strategy, and planning) and introduce a hierarchical failure-routing architecture with a Judger-led direct path that efficiently resolves manageable queries while routing genuinely complex ones to structured decomposition.
- We introduce a plan-level repair mechanism that identifies persistent query failures as signs of a flawed decomposition, automatically restructuring the reasoning DAG in response. Evaluating on the BIRD and Spider benchmarks, DecoSearch achieves execution accuracies of 70.53% and 88.31%, respectively. Without any supervised fine-tuning, it successfully recovers a significant portion of the queries where direct generation baselines fail.

2 Related Work

Text-to-SQL with LLMs. Early Text-to-SQL systems relied on semantic parsing and schema-aware neural encoders (Wang et al., 2020; Cai and Wan, 2020). The rise of large-scale LLMs shifted the paradigm toward in-context learning, enabling strong zero-shot and few-shot performance (Rajkumar et al., 2022; Gao et al., 2024). However, single-pass generation remains brittle on complex queries requiring multi-table joins or data-aware reasoning (Li et al., 2023, 2024), a challenge made even more acute by newer, harder benchmarks such as Spider 2.0 (Lei et al., 2025), which features enterprise-scale databases where state-of-the-art models achieve below 25% accuracy.

Decomposition-based Methods. A natural response to this brittleness is to decompose complex questions into simpler sub-problems (Zhou et al., 2023; Pourreza and Rafiei, 2023; Eyal et al., 2023; Wang et al., 2025a). DIN-SQL (Pourreza and Rafiei, 2023) pioneered decomposed in-context learning with self-correction for Text-to-SQL. DEA-SQL (Xie et al., 2024) introduces a workflow-based paradigm that counters attention diffusion in monolithic prompting through sequential filtering, classification, and correction stages. Our work extends these ideas by formalizing the decomposition as a dependency DAG and enabling plan-level repair when sub-problems fail.

Schema Linking. Passing the full database schema to every LLM call degrades generation quality and inflates cost. CHESS (Talaei et al., 2024) integrates LLM-driven schema pruning as a dedicated agent. RSL-SQL (Cao et al., 2024) further improves schema linking with bidirectional pruning, achieving high recall while cutting input columns substantially. DecoSearch's schema selector follows this paradigm, applying LLM-driven keyword extraction and two-stage table and column selection before any generation call.

Self-Refinement and Execution Feedback. Several methods improve robustness by iteratively revis-

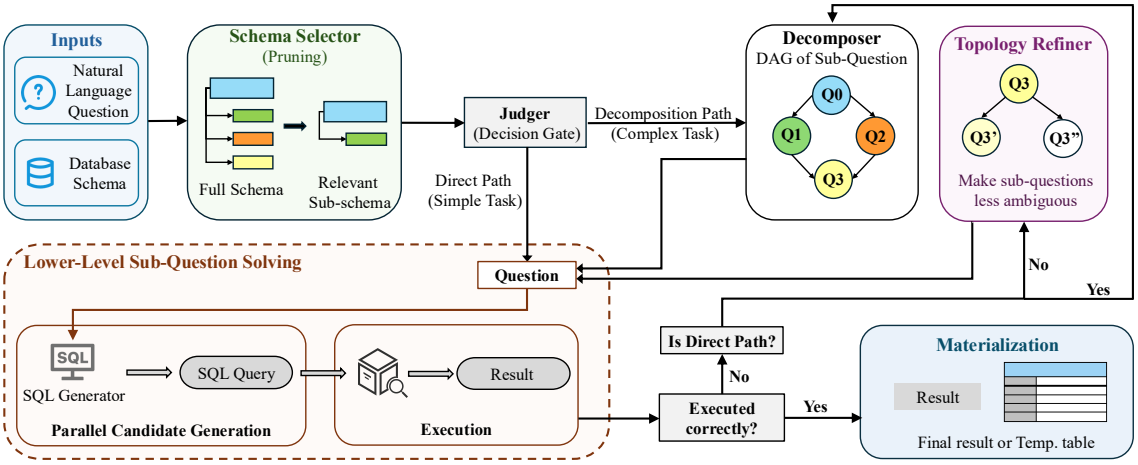


Figure 1: The pipeline begins with Pre-Processing via evidence-aware schema pruning. A Judger determines if the question is manageable; if so, the Direct Path generates and executes a single SQL query and exits if successful. Otherwise, the Decomposer produces a DAG of atomic sub-questions linked by `@[n]` dependency placeholders. At the Sub-Question Solving stage, each node is executed in topological order; upstream results are resolved as inline value lists for small result sets or materialized as temporary tables for large ones before the downstream SQL is generated. Persistent execution failures trigger the Topology Refiner to rewrite the DAG and retry.

ing SQL using execution feedback (Madaan et al., 2023; Gou et al., 2024; Chen et al., 2024; Askari et al., 2025). ExCoT (Zhai et al., 2025) extends this with execution-supervised fine-tuning, iteratively improving open-source models using only SQL execution accuracy as the reward signal. These approaches address implementation-level failures but cannot restructure an incorrect reasoning plan. DecoSearch complements this by operating at the plan level: when execution feedback reveals a flawed decomposition, the system rewrites the DAG rather than patching individual queries.

Multi-Candidate, Routing, and Test-Time Scaling. CHASE-SQL (Pourreza et al., 2024) generates diverse candidates via multiple CoT strategies and selects the best via a fine-tuned pairwise ranker. Agentar-Scale-SQL (Wang et al., 2025b) combines RL, refinement, and parallel synthesis at the query level. CHESS (Talaei et al., 2024) and MCS-SQL (Lee et al., 2025) use self-consistency voting over multiple candidates; Query and Conquer (Borchmann and Wydmuch, 2025) shows that execution-guided selection alone enables smaller models to match much larger ones. Complexity-aware routing has also emerged as a practical tool: EllieSQL (Zhu et al., 2025) routes queries to different generation pipelines based on estimated difficulty, achieving substantial token savings, while Rethinking Agentic Workflows (Guo et al., 2025) benchmarks test-time scaling strategies for Text-to-SQL and finds that routing and decomposition consistently outperform naive candidate scaling. DecoSearch uses a Judger-led routing strategy to select between direct generation and hierarchical decomposition, reserving the structured DAG approach for queries requiring deeper reasoning.

3 Methodology

3.1 Problem Formulation

Given a natural language question Q, a database $\mathcal{D}$ with schema $\mathcal{S}$, and optionally a set of domain hints ("evidence"), our goal is to produce an SQL query $\hat{Q}$ such that $\text{Execute}(\hat{Q}, \mathcal{D}) = \text{Execute}(Q^*, \mathcal{D})$, where Q^* is the ground-truth query. We evaluate correctness via execution accuracy (EX): a query is correct if and only if its result set semantically matches the gold result set.

3.2 Overview

DecoSearch treats Text-to-SQL generation as an adaptive, multi-level reasoning process. Rather than committing every question to the same generation procedure, the system first assesses query complexity and routes each question to the cheapest path that is likely to succeed. For straightforward questions, a direct generation attempt is sufficient. For complex questions, those requiring multi-table joins, nested logic, or multi-step inference; the system escalates to a structured decomposition that breaks the question into a DAG of simpler, inde-

pendently solvable sub-questions. Failures are diagnosed and repaired at the appropriate level of abstraction: when a node's SQL fails to execute, the system signals a flawed reasoning plan and triggers topology refinement of the DAG. The entire framework is training-free and operates at test-time, relying solely on the reasoning capabilities of an off-the-shelf LLM.

Our proposed framework consists of five primary components: (1) a Schema Selector that prunes the database schema to the most relevant tables before any LLM call, (2) a Judger-Led Escalation strategy that determines the appropriate reasoning depth, (3) a Direct Path for manageable questions, (4) a Query-Level SQL Generation step for sub-problem solving, and (5) a Hierarchical Control Loop for dynamic topology refinement. This tiered approach ensures that computational resources are allocated proportionally to the complexity of the question, with decomposition reserved for queries the Judger identifies as complex. The full pipeline is given in Figure 1. We describe each component below.

3.3 Schema Selection

Real-world databases often contain dozens of tables and hundreds of columns, the vast majority of which are irrelevant to any given question. Passing the full schema to every LLM call wastes tokens, inflates cost and, more critically, degrades generation quality by filling the context with distracting schema noise. DecoSearch therefore applies a lightweight schema pruning step before any LLM call in the pipeline.

Our Schema Selector follows the schema-pruning approach introduced in CHESS (Talaei et al., 2024) and DeepEye-SQL (Li et al., 2026), and is consistent with the bidirectional pruning principles of RSL-SQL (Cao et al., 2024), adapting their keyword-based filtering strategy to our pipeline. The Schema Selector operates in two stages. First, it issues a single LLM call to extract a set of task-relevant keywords and entities from the NLQ and the evidence string (e.g., column names, value filters, domain terms). Second, it uses those keywords to drive an LLM-based table selection call, followed by a column selection call within the chosen tables. Only the selected tables and their columns are serialized into the schema string passed to the Judger, Decomposer, SQL Generator, and Topology Refiner.

This pruning reduces prompt length by up to an order of magnitude for large databases and focuses the LLM's attention on relevant schema elements; the ablation study (§4.3) confirms its impact on both accuracy and token cost.

3.4 Judger-Led Escalation and Direct Path

The pipeline begins with a specialized LLM Judger that evaluates the complexity of the natural language question (NLQ) relative to the database schema. The Judger is a single zero-shot LLM call that receives two inputs: (1) the NLQ and (2) the pruned database schema produced by the preceding schema-selection step. It outputs a structured JSON object of the form `{"needs_decomposition": true/false, "reasoning": "..."}`, providing both a binary routing decision and a brief rationale.

The Judger prefers the direct path by default, escalating to decomposition only for questions with conflicting aggregation grains, extreme join complexity, multi-phase reasoning where step two depends on a materialized result from step one, or evidence-driven formulas that would make a single query excessively nested. This routing philosophy is consistent with recent work on complexity-aware allocation (Zhu et al., 2025; Guo et al., 2025), which shows that matching computational effort to query difficulty yields significant accuracy and efficiency gains over uniform strategies.

When decomposition is not needed, the pipeline proceeds directly to the Direct Path described below; when it is, the system skips the direct path and proceeds to DAG-based decomposition.

For questions judged as manageable, the system follows a Direct Path: a single SQL query is generated using a zero-temperature LLM call and executed against the database. If execution succeeds, the result is returned immediately, keeping both latency and cost low. If execution fails, the framework falls through to the decomposition stage, which systematically breaks the original query into a sequence of simpler, more verifiable sub-tasks.

3.5 Plan-Level Search: Task Decomposition

When the Judger mandates escalation or the direct path fails to execute, the system initiates an upper-level plan by decomposing the complex NLQ into a DAG. In this graph, nodes represent simpler, atomic sub-questions, and directed edges represent data dependencies between them. The topological order of the graph determines the execution sequence, ensuring that each node's inputs are fully materialized before it executes.

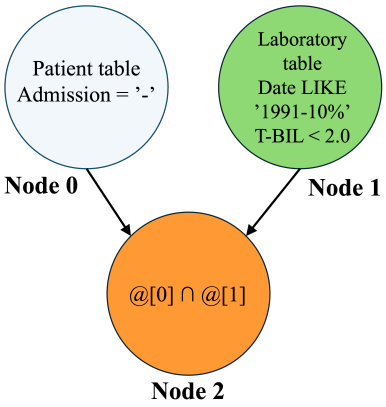


Figure 2: DAG for the `thrombosis_prediction` example. Node 2 executes last, injecting results of Nodes 0 and 1 via `@[n]` placeholders.

Example 1. Consider the following question from the BIRD benchmark:

> *"List all patients who were followed up at the outpatient clinic who underwent a laboratory test in October 1991 and had a total blood bilirubin level within the normal range."*

The Decomposer breaks this into the three-node DAG shown in Figure 2. Nodes 0 and 1 are independent and can be solved in parallel; Node 2 consumes both their results. The `@[n]` syntax is a dependency placeholder: at execution time, the result set of Node n is resolved and injected into the downstream node's prompt and SQL before generation. For small result sets ($\leq$100 rows), values are inlined as a SQL value list (e.g., an `IN (...)` literal); for larger result sets, the upstream result is materialized into a SQLite temporary table (`temp_ds_node_n`), which the downstream node can query by name. This design prevents the LLM from having to express the full multi-table logic in a single query while remaining efficient regardless of result cardinality (see Appendix A.3 for details).

To improve the logical soundness of this initial decomposition, we employ a Retrieval-Augmented Generation (RAG) strategy. Before generating the graph, the system retrieves a set of successful question-decomposition pairs from a pre-populated knowledge base. These examples are dynamically injected into the decomposer's prompt as few-shot demonstrations, grounding the LLM in relevant, structured examples and guiding it toward more coherent and viable task graphs.

The knowledge base is constructed offline from prior successful runs on the benchmark's training split: experiment logs are parsed to extract question–sub-question pairs for which the pipeline produced a correct final answer, and these are stored as a JSONL file. At inference time, we encode the incoming question using a sentence embedding model (Reimers and Gurevych, 2019) and retrieve the $k = 3$ most similar training examples by cosine similarity.

3.6 Query-Level SQL Generation

For each node in the decomposition graph, the system generates a single SQL query using a zero-temperature LLM call (full prompt in Appendix C.3). The prompt instructs the model to produce a valid SQL query grounded in the pruned schema and any available evidence hints, while applying standard robustness conventions such as case-insensitive string comparisons. The generated query is executed against the database; if it succeeds, its result is materialized and passed to downstream nodes via the `@[n]` placeholder mechanism. If execution fails, the node signals failure to the Hierarchical Control loop, which then decides whether to trigger topology refinement. The full pipeline is described in Algorithm 1.

3.7 Hierarchical Control: Dynamic Topology Refinement

To improve resilience against flawed initial decompositions, DecoSearch maintains a hierarchical feedback loop that enables dynamic plan correction at the graph level. This mechanism allows the system to recover from cases where the decomposition strategy itself is the root cause of failure, rather than individual SQL generation.

Trigger Condition. Topology refinement is triggered when the generated SQL at a given node fails to execute successfully. This signals that the sub-question may be ill-formed, overly broad, or structurally inconsistent with the available schema; problems that cannot be resolved by regenerating a query for the same sub-question.

Action and Resumption. Upon receiving the failure signal, the framework activates the Topology Refiner module that provides the LLM with the current DAG structure, the failing node's sub-question, and the execution error from the failed query. The LLM's task is to produce a revised

DAG, which may involve re-partitioning dependencies, inserting intermediate nodes to break down an overly complex sub-question, or restructuring the data-flow to better align with the schema. The pipeline then adopts the revised graph and resumes execution from the first modified node, preserving the results of all unaffected upstream nodes. Refinement is bounded by a budget B (the maximum number of topology refinement attempts per query). If the revised DAG also fails to execute and the budget is exhausted, the system accepts the best available result for the failing node and continues execution of all downstream nodes. This graceful degradation ensures that a single irresolvable sub-problem does not abort the entire pipeline.

4 Experiments

We conduct empirical evaluations to assess the performance of our proposed framework. The experiments are designed to answer three primary research questions: (1) How does our hierarchical search framework compare against state-of-the-art (SOTA) Text-to-SQL models and (2) how much does this approach reduce token usage and cost compared to standard methods? (3) Can the proposed method be combined with fine-tune models to improve accuracy?

Algorithm 1 DecoSearch Pipeline

Require: Natural Language Question Q, Database $\mathcal{D}$, Schema $\mathcal{S}$
Ensure: Final SQL Result $\mathcal{R}$

{**Step 1: Schema Selection**}
$\mathcal{S}_{pruned} \leftarrow$ SchemaSelector$(Q, \mathcal{S})$
{**Step 2: Judger — Direct Path or Decompose**}
if Judger$(Q, \mathcal{S}_{pruned})$ is Direct Path **then**
  $\hat{Q} \leftarrow$ SQLGenerator$(Q, \mathcal{S}_{pruned})$
  R, Error $\leftarrow$ Execute$(\mathcal{D}, \hat{Q})$
  if Error $=$ NONE **then**
    return R {Direct execution succeeded}
  end if
  {Execution failed: fall through to decomposition}
end if
{**Step 3: Decomposition**}
$G \leftarrow$ Decomposer$(Q, \mathcal{S}_{pruned})$ {DAG of sub-questions}

$Queue \leftarrow$ TopologicalSort(G)
$Results \leftarrow \{\}$
{**Step 4: Hierarchical Control & Materialization**}
while $Queue$ is not empty **do**
  $n \leftarrow Queue$.pop(0)
  $Q_{sub} \leftarrow$ Substitute$(n.question, Results)$
  $\hat{Q}_{sub} \leftarrow$ SQLGenerator$(Q_{sub}, \mathcal{S}_{pruned})$
  R_{sub}, Error $\leftarrow$ Execute$(\mathcal{D}, \hat{Q}_{sub})$
  if Error $\neq$ NONE **then**
    $G \leftarrow$ TopologyRefiner$(G, n,$ Error$)$
    if G is revised **then**
      $Queue \leftarrow$ TopologicalSort(G)
      continue
    end if
  end if
  $Results[n] \leftarrow$ Materialize$(\mathcal{D}, \hat{Q}_{sub}, R_{sub})$
end while
return $Results$[final]

Datasets. We evaluate on two standard benchmarks. Spider (Yu et al., 2018) contains 2,147 testing queries across 200 databases. It emphasizes structural SQL complexity (multi-table joins, nested queries). BIRD (Li et al., 2023) contains 1,534 development queries across 11 databases designed to test "data-aware" reasoning, requiring the model to understand actual database content (e.g., value distributions, cell-level patterns).

4.1 Experimental Setup

Our experimental setup is defined as follows.

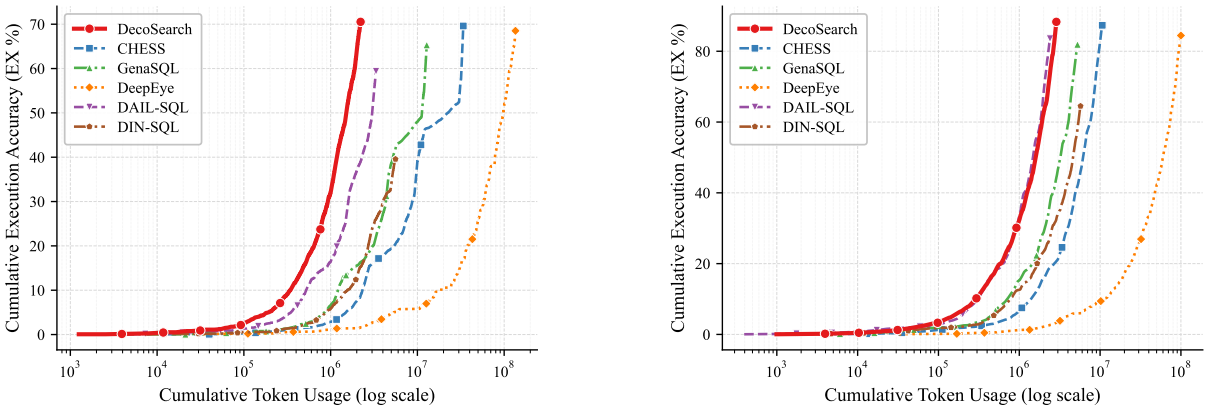
Baselines. Our method is tested with two categories of baselines: Training-Free and Fine-tuned methods. Training-free methods operate solely via prompting without any task-specific fine-tuning, including CHESS (Talaei et al., 2024), GenaSQL (Dönder et al., 2025), and DAIL-SQL (Gao et al., 2024). Fine-tuned methods additionally train or adapt model weights on Text-to-SQL data, including XiYan-SQL (Liu et al., 2026), Agentar-Scale-SQL (Wang et al., 2025b), CHASE-SQL (Pourreza et al., 2024), and OmniSQL (Li et al., 2025). Because our approach focuses on test-time reasoning strategy, query routing, structured decomposition, and plan-level repair, rather than on model parameters or training data, it is orthogonal to these fine-tuned methods and can be readily combined with them to yield further improvements.

Implementation Details. Our framework is model-agnostic. For all experiments reported in this paper, all LLM-driven components (Judger, Decomposer, SQL Generator, Topology Refiner) use DeepSeek as the backbone model with a single greedy generation per node. All LLM calls are cached to ensure full reproducibility. SQL Generator prompts are lightly adapted per dataset to reflect dataset-specific conventions: the BIRD prompt enforces case-insensitive string comparisons (via `LOWER()`) and treats the evidence string as ground truth, while the Spider prompt emphasizes column examples present in the schema. The core reasoning structure is identical across both. Full hyperparameter settings are reported in Appendix B.

Evaluation Metric. The primary metric is Execution Accuracy. A generated SQL query is deemed

Table 1: Comprehensive comparison of Text-to-SQL frameworks on BIRD and Spider benchmarks, categorized by learning approach. "Model Avail." indicates whether model weights are publicly available.

Learning Approach	Method	Model Avail.	BIRD EX (%) dev	Spider EX (%) dev
Training-free	DeepEye-SQL	×	68.51	84.44
	GenaSQL	×	65.32	81.93
	CHESS	×	69.60	87.33
	DAIL-SQL	×	59.39	83.56
	DIN-SQL	×	39.57	64.51
Open-source	GPT-OSS-120b	✓	66.16	84.16
	GPT-OSS-20b	✓	64.21	83.09
	Llama-4-Maverick-17B	✓	67.27	85.42
	Mistral-nemotron-12B	✓	61.73	81.93
	Qwen2.5-7B	✓	39.96	77.27
	DeepSeek	–	66.81	85.56
	DecoSearch (ours)	–	**70.53**	**88.31**



(a) BIRD (b) Spider

Figure 3: Cumulative Execution Accuracy vs. Cumulative Token Usage (log scale) on BIRD (a) and Spider (b). DecoSearch reaches its final accuracy with fewer tokens than all compared methods on both benchmarks, demonstrating superior token efficiency.

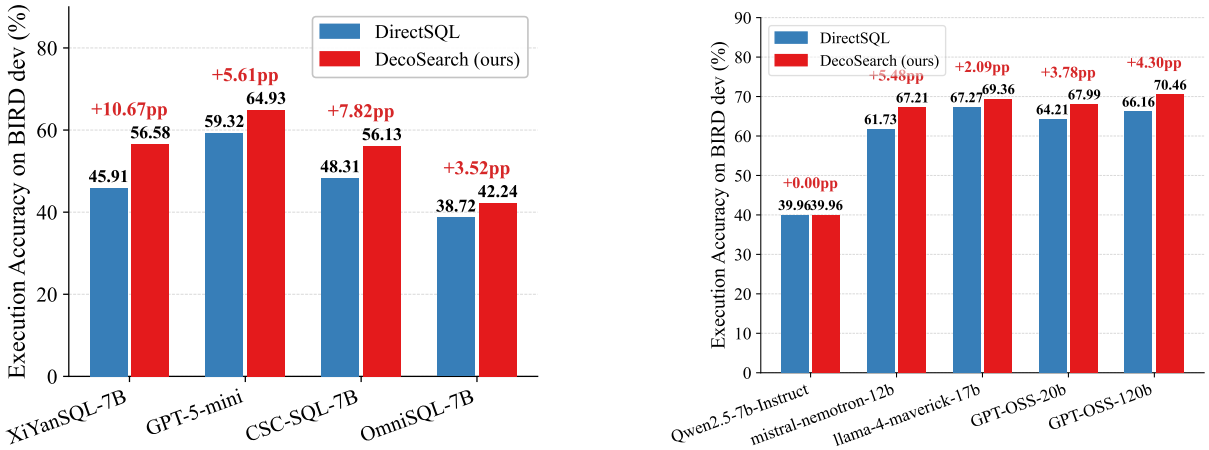
correct if and only if its execution result set is identical to that of the ground-truth query. Result sets are canonicalized to be robust against arbitrary row and column ordering, as well as known surface-level formatting artifacts (e.g., concatenated vs. separate string tokens that convey the same information).

4.2 Results and Analysis

Overall Performance. Evaluation results are summarized in Table 1. On BIRD and Spider, DecoSearch achieves 70.53% and 88.31% EX, respectively, consistently outperforming all training-free baselines. These findings validate that our primary performance gains stem from the hierarchical decomposition and the Judger-led routing strategy.

Token Efficiency. Figure 3 plots cumulative EX accuracy against cumulative token usage (log scale) for all methods. DecoSearch is consistently the leftmost high-accuracy curve on both benchmarks, demonstrating that it reaches its peak accuracy with fewer tokens than every compared method. On BIRD, DecoSearch converges near 2×10^6 tokens; CHESS requires approximately 2×10^7 tokens to reach a comparable accuracy, a $10\times$ overhead. On Spider, DecoSearch achieves 88.31% at roughly 2×10^6 tokens, a budget at which most baselines remain below 60% cumulative accuracy. This efficiency advantage is a direct consequence of the Judger-Led Escalation strategy: the majority of queries are resolved via the direct path, and the expensive decomposition machinery is invoked only for the fraction of queries that genuinely require it.

DecoSearch as a General-Purpose Wrapper for Fine-Tuned Models. A key property of DecoSearch is that its hierarchical search framework is entirely model-agnostic: the backbone LLM can be swapped without any modification to the pipeline. To demonstrate this, we pair DecoSearch with several SOTA fine-tuned SQL generation models and evaluate on BIRD. Figure 4 shows that



(a) Fine-tuned backbones

(b) Open-source backbones

Figure 4: Execution Accuracy on BIRD when pairing DecoSearch with fine-tuned (a) and open-source (b) backbone models, compared against their DirectSQL baseline. DecoSearch consistently improves over the direct generation baseline regardless of the backbone, demonstrating its model-agnostic nature.

DecoSearch consistently improves over the corresponding DirectSQL baseline for every backbone, with gains of +10.67 pp (XiYanSQL-7B), +5.61 pp (GPT-5-mini), and +3.52 pp (OmniSQL-7B), where pp denotes percentage points. These results confirm that DecoSearch's decomposition and routing strategy is complementary to model-level fine-tuning: it provides an additional layer of structured reasoning on top of whatever SQL generation capability the backbone already possesses, and can therefore be applied as a drop-in enhancement to any existing fine-tuned system.

4.3 Ablation Study

To quantify the contribution of each component, we conduct an ablation study on a 770-query subset of BIRD, using DeepSeek as the backbone, shown in Table 2. The Judger is the dominant component: forcing all queries through decomposition causes the largest drop (−10.65 pp), revealing that DAG decomposition actively hurts simple queries. The Judger's role is to prevent from over-engineering. The schema selector is the second most impactful component (−1.82 pp), while also inflating token cost 6.3× when removed. Removing decomposition entirely costs only −1.43 pp, confirming it targets a small but consequential subset. Topology refinement provides an additional incremental gain (−0.78 pp), with the primary benefit coming from routing and hierarchical decomposition.

Table 2: Ablation study on BIRD (dev). Each row removes one component from the full DecoSearch pipeline. Δ EX is the absolute change in Execution Accuracy relative to the full system.

Configuration	**EX (%)**	**Δ EX**	**Tokens** ($\times 10^6$)
Full Pipeline	69.48	—	2.9
w/o Judger (always decompose)	58.83	−10.65	2.0
w/o Decomposition (direct path only)	68.05	−1.43	1.8
w/o Topology Refinement	68.70	−0.78	2.5
w/o Schema Selector	67.66	−1.82	18.1

5 Conclusion

We presented DecoSearch, a training-free framework that adaptively matches computational effort to query difficulty by routing each question through a Judger-Led Escalation strategy: simple queries are resolved via direct generation, while complex ones are escalated to a DAG decomposition process with automatic Topology Refinement. DecoSearch achieves 70.53% execution accuracy on BIRD and 88.31% on Spider, surpassing all training-free baselines and outperforming several fine-tuned systems on Spider, while consuming an order of magnitude fewer tokens than methods such as CHESS. The ablation study confirms that Judger-led routing is the dominant contributor (−10.65 pp when disabled), underscoring that DecoSearch's value lies in the principled orchestration of routing, decomposition, and plan-level repair.

6 Limitations

DecoSearch relies on executable database feedback during inference. The topology refiner is triggered

by SQL execution errors at individual DAG nodes, which assumes that queries can be executed against the target database and that execution failures are informative signals of plan level flaws. This assumption is common across most training free Text-to-SQL methods that incorporate self-refinement or candidate selection, since execution feedback is the strongest unsupervised signal available at test time. For settings where execution is unavailable, prohibitively expensive, or returns semantically wrong but syntactically valid results, DecoSearch, like other execution guided methods, cannot intervene. Nevertheless, the framework remains applicable to any setting where SQL can be executed against the target database, which covers all standard Text to SQL benchmarks and the majority of deployed analytical workloads.

References

Arian Askari, Christian Poelitz, and Xinye Tang. 2025. MAGIC: Generating Self-Correction Guideline for In-Context Text-to-SQL. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Łukasz Borchmann and Marek Wydmuch. 2025. Query and Conquer: Execution-Guided SQL Generation. *arXiv preprint arXiv:2503.24364*.

Yitao Cai and Xiaojun Wan. 2020. IGSQL: Database Schema Interaction Graph Based Neural Model for Context-Dependent Text-to-SQL Generation. In *2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. RSL-SQL: Robust Schema Linking in Text-to-SQL Generation. *arXiv preprint arXiv:2411.00073*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *International Conference on Learning Representations (ICLR)*.

Yusuf Denizay Dönder, Derek Hommel, Andrea W Wen-Yi, David Mimno, and Eun Seo Jo. 2025. Cheaper, Better, Faster, Stronger: Robust Text-to-SQL without Chain-of-Thought or Fine-Tuning. *arXiv preprint arXiv:2505.14174*.

Ben Eyal, Moran Mahabi, Ophir Haroche, Amir Bachar, and Michael Elhadad. 2023. Semantic Decomposition of Question and SQL for Text-to-SQL Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

Dawei Gao, Haibin Wang, Yaliang Li, and 1 others. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. In *Proceedings of the VLDB Endowment*, volume 17.

Zhibin Gou, Zhihong Shao, Yeyun Gong, and 1 others. 2024. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*.

Jiajing Guo, Kenil Patel, Jorge Piazentin Ono, Wenbin He, and Liu Ren. 2025. Rethinking Agentic Workflows: Evaluating Inference-Based Test-Time Scaling Strategies in Text-to-SQL Tasks. *arXiv preprint arXiv:2510.10885*.

Arthur B. Kahn. 1962. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

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

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection for Text-to-SQL Generation. *International Conference on Computational Linguistics (COLING)*.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, and 1 others. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *International Conference on Learning Representations (ICLR)*.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2026. DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework. In *ACM Special Interest Group on Management of Data (SIGMOD)*.

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.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing high-quality text-to-SQL data at scale. *Proceedings of the VLDB Endowment*. ArXiv:2503.02240.

Jinyang Li, Binyuan Hui, Ge Qu, and 1 others. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems*, volume 36.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026. XiYan-SQL: A Novel Multi-Generator Framework For Text-to-SQL. *arXiv preprint arXiv:2507.04701*.

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):5466–33547131.

Aman Madaan, Niket Tandon, Pranjal Gupta, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

Mohammadreza Pourreza, Hailong Chang, Nima Amin, and 1 others. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the Text-to-SQL Capabilities of Large Language Models. *arXiv preprint arXiv:2204.00498*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Ruoxi Sun, Sercan Arik, Alex Muzio, Lesly Miculivicius, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *arXiv preprint arXiv:2306.00739*.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, and 1 others. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. In *arXiv preprint arXiv:2405.16755*.

Bailin Wang, Richard Shin, Xiaodong Liu, and 1 others. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2025a. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *International Conference on Computational Linguistics (COLING 2025)*.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025b. Agentar-Scale-SQL: Advancing Text-to-SQL through Orchestrated Test-Time Scaling. *arXiv preprint arXiv:2509.24403*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, and 1 others. 2023. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, MingXiong Lin, Liang Chen, and 1 others. 2024. Decomposition for Enhancing Attention: Improving LLM-based Text-to-SQL through Workflow Paradigm. *arXiv preprint arXiv:2402.10671*.

Tao Yu, Rui Zhang, Kai Yang, and 1 others. 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*.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. ExCoT: Optimizing Reasoning for Text-to-SQL with Execution Feedback. *arXiv preprint arXiv:2503.19988*.

Denny Zhou, Nathanael Schärli, Le Hou, and 1 others. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *International Conference on Learning Representations (ICLR)*.

Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. EllieSQL: Cost-Efficient Text-to-SQL with Complexity-Aware Routing. *arXiv preprint arXiv:2503.22402*.

A Detailed Algorithms

A.1 Schema Selector

Algorithm 2 details the SCHEMASELECTOR component that prunes the full database schema before any downstream LLM call. The selector makes three sequential LLM calls: first to extract task-relevant keywords from the question and evidence, then to identify the necessary tables, and finally to pinpoint the required columns within those tables. All LLM-returned names are remapped to their true schema counterparts via exact match followed by fuzzy match (FM), guarding against surface-level naming discrepancies in the LLM output. FUZZYMATCH finds the closest LLM-returned name to each true schema name by string similarity, returning $\varnothing$ if no sufficiently close match exists. If the pruning procedure would produce an empty schema, the original schema is returned instead.

A.2 Topological Execution Order

Algorithm 3 computes the node execution order from the decomposition DAG using Kahn's algorithm (Kahn, 1962). In the DAG $G = (V, E)$, a directed edge (u, v) indicates that node v consumes the result of node u, so u must execute before v.

Algorithm 2 SCHEMASELECTOR: LLM-Driven Schema Pruning

Require: Question Q, full schema $\mathcal{S}$, evidence E
Ensure: Pruned schema $\mathcal{S}_{pruned}$, keyword list K

{**Step 1: Keyword Extraction**}
$K \leftarrow \text{LLM}_{\text{IR}}(Q, E)$ {Extract keywords, entities, value mappings}
{**Step 2: Table Selection**}
$\mathcal{T}_{sel} \leftarrow \text{LLM}_{\text{TS}}(Q, E, K, \mathcal{S})$ {Select relevant table names}
{**Step 3: Column Selection**}
$\mathcal{C}_{sel} \leftarrow \text{LLM}_{\text{CS}}(Q, E, K, \mathcal{T}_{sel})$ {Select relevant columns per table}
{**Step 4: Name Remapping & Schema Assembly**}
$\mathcal{S}_{pruned} \leftarrow \{\}$
for all table $t \in \mathcal{S}$ **do**
  $\hat{t} \leftarrow \text{FM}(t.name,\ \mathcal{T}_{sel})$
  if $\hat{t}$ found **then**
    $C_t \leftarrow \{c \in t.cols : \text{FM}(c.name,\ \mathcal{C}_{sel}[\hat{t}])\}$
    if $C_t \neq \emptyset$ **then**
      $\mathcal{S}_{pruned}.\text{add}(t.name,\ C_t)$
    end if
  end if
end for
if $\mathcal{S}_{pruned} = \emptyset$ **then**
  return $\mathcal{S},\ K$ {Fallback: return original schema}
end if
return $\mathcal{S}_{pruned},\ K$

Leaf nodes (in-degree zero) have no upstream dependencies and are scheduled first; a cycle detection check guards against malformed LLM outputs.

Algorithm 3 TOPOLOGICALSORT: Dependency-Aware Execution Ordering

Require: DAG $G = (V, E)$; edge (u, v) means v depends on u
Ensure: Execution queue $Queue$ with all dependencies before dependents

$depCount[v] \leftarrow |\{u : (u, v) \in E\}|$ for all $v \in V$
$Ready \leftarrow \{v \in V : depCount[v] = 0\}$ {Leaf nodes}
$Queue \leftarrow [\,]$
while $Ready$ is not empty **do**
  $n \leftarrow Ready.\text{pop}()$
  $Queue.\text{append}(n)$
  for all dependent d such that $(n, d) \in E$ **do**
    $depCount[d] \leftarrow depCount[d] - 1$
    if $depCount[d] = 0$ **then**
      $Ready.\text{add}(d)$ {All of d's dependencies resolved}
    end if
  end for
end while
if $|Queue| < |V|$ **then**
  raise CYCLEDETECTEDERROR
end if
return $Queue$

A.3 Dependency Placeholder Resolution

When a downstream node depends on the result of an upstream node u via the `@[`u`]` placeholder, DecoSearch must inject that result into the downstream node's sub-question and SQL prompt before generation begins. The injection strategy is chosen based on the cardinality of node u's result set.

Small result sets ($|\mathcal{R}_u| \leq \theta$). If the result contains at most $\theta = 100$ rows, the values are serialized directly as a SQL value list. For example, `@[1]` in the sub-question text is replaced with the literal tuple `('v1', 'v2', ...)`, which can appear directly inside an `IN (...)` clause in the generated SQL.

Large result sets ($|\mathcal{R}_u| > \theta$). When the result exceeds θ rows, inlining the values would produce an unwieldy prompt and an oversized SQL literal. Instead, DecoSearch materializes the result into a *SQLite temporary table* using:

`CREATE TEMPORARY TABLE temp_ds_node_`u `AS` $\langle \texttt{SQL}_u \rangle$

The temporary table `temp_ds_node_`u is then referenced by name wherever `@[`u`]` appears in downstream SQL, and a schema description comment is prepended to the downstream node's prompt so the LLM knows the table exists and can query it like any ordinary table. The temporary table persists for the lifetime of the database connection and is therefore available to all subsequent nodes in the same execution run.

Fallback. If temporary-table creation is disabled or the result is empty, values are truncated to a adjustable limit (200 rows by default) with a comment appended to the SQL noting the truncation.

A.4 Topology Refiner

Algorithm 4 details TOPOLOGYREFINER, invoked from the main pipeline (Algorithm 1) when execution fails at a DAG node and the refinement budget B has not been exhausted. The refiner serializes the current graph to JSON, constructs a prompt that includes the failing node's sub-question and the execution error, and asks the LLM to produce a structurally revised DAG. The caller rejects the revision if it is isomorphic to the original (no structural change) or contains a cycle, and falls back to the original graph in either case.

B Detailed Experiment Setup

All experiments on both BIRD and Spider use the same configuration unless otherwise stated. DeepSeek-V3 (`deepseek-chat`) serves as the backbone for every LLM role in the pipeline, and

Algorithm 4 TOPOLOGYREFINER: LLM-Driven DAG Revision

Require: DAG G, failing node n^*, failure reason ρ, question Q, evidence E
Ensure: Revised DAG G' (or G if revision is invalid)

```
J ← Serialize(G) {Graph → JSON}
prompt ← BuildPrompt(Q, E, J, n*, ρ)
response ← LLM(prompt)
J' ← ExtractJSON(response)
if J' is valid JSON then
    G' ← Deserialize(J')
    if G' is isomorphic to G then
        return G {No structural change; skip}
    end if
    if G' contains a cycle then
        return G {Reject invalid DAG}
    end if
    return G'
else
    return G {Parse failure; retain original}
end if
```

a global random seed of 42 is fixed across all API calls to ensure reproducibility. The judger and the direct SQL generator both run at temperature $T = 0$ to produce deterministic outputs, while the decomposer runs at $T = 0.1$ to allow for slightly varied but still constrained decomposition strategies. The RAG-based decomposer retrieves $k = 3$ similar examples from the training knowledge base using `all-MiniLM-L6-v2` embeddings to construct few-shot decomposition prompts. If execution fails at any DAG node, the topology refiner is invoked for up to $B = 3$ revision attempts before falling back to the best available result. The schema selector runs a three-stage LLM pipeline (keyword extraction, table selection, column selection) to prune the full database schema before any downstream call; pruning is adaptive with no hard cap on the number of retained tables, and fuzzy matching is applied to guard against naming discrepancies in the LLM output.

C Prompts

We report the core prompt templates used by each pipeline component. Slots filled dynamically at runtime are shown in `[SLOT]`.

C.1 Judger Prompt

Judger Prompt

System: You are an expert SQL architect and query planner. Your job is to analyze a natural language question and a database schema to decide if the question should be decomposed into multiple sub-questions or if it can be solved reliably with a single direct SQL query.

STRATEGY: PREFER DIRECT SQL
Empirical analysis shows that our Direct SQL path is highly accurate. Decomposition should be used ONLY as a "Heavy Lifting" tool for logic that cannot be reliably expressed in a single pass.

1. When to choose Direct SQL (needs_decomposition: false):
- Standard Relational Lookups: Even with 2-4 joins, a single query is usually best.
- Simple Filters & Mappings: Using values from the Hint to filter columns.
- Standard Aggregations: SUM, AVG, COUNT, MAX/MIN per group.
- Basic Subqueries: WHERE id IN (SELECT ...) or simple CTEs.

2. When to choose Decomposition (needs_decomposition: true):
- **Conflicting Aggregation Grains:** Asking for a detail and a scalar at once (e.g., "List all students AND the average score").
- **Extreme Join Complexity:** Joining $\geq$5 tables where the path is highly ambiguous.
- **Complex Multi-Phase Reasoning:** When Step 2 depends on a mathematical result from Step 1 that must be calculated first (e.g., "Find the top earner, then find all their transactions").
- **Evidence-Driven Fork:** If the Hint specifies a formula or complex mapping that makes a single query extremely nested and error-prone.

Input:
Database Schema: {schema}
Natural Language Question: {question}
Task: Your task is to analyze the question and schema. Decide if the question NEEDS to be decomposed.

Output: Respond with a JSON object. Only output the JSON object. Do not include any other text or markdown formatting.
{"needs_decomposition": true/false, "reasoning": "A brief explanation of your decision based on the criteria (simplicity, JOINs, logical steps)."}

Now, take a deep breath, and think step by step to make your decision.

C.2 Decomposer Prompt

Decomposition Planner Prompt

System: You are a specialized SQL Architect and Query Planner. Your task is to decompose a complex natural language question into a logically sound dependency graph (DAG) of simpler sub-questions.

MISSION: Break the question into atomic, executable steps where each step produces a result required by the next.

STRICT DECOMPOSITION RULES:
- **1. MANDATORY IDENTIFIER PASSING:** If Step A depends on Step B, Step B MUST return a

specific identifier (like an ID, Code, or Name) that Step A can use for filtering. Example: "Find the IDs of..." followed by "Find the details for IDs in @[ID]".

- **2. PLACEHOLDER SYNTAX:** Use @[node_id] to reference the result of a previous step.
- **3. EVIDENCE IS THE MAP:** BIRD Evidence often contains formulas or value mappings. If the evidence says "'Active' means status=1", create a sub-question that handles that specific filter or calculation.
- **4. NO FRAGMENTATION:** Do not create a new node for every single clause. Only decompose if the result of one query is logically required to build the next (e.g., Finding a Max/Min, or finding a set of IDs to filter a different table).
- **5. ROOT NODE PRECISION:** The root node (ID 0) must be the final transformation. It should return the exact column(s) requested in the original question using the results from its children.

Use the provided successful examples below to understand the correct decomposition format and style.

{examples_prompt_str}

Input:
Database Schema: {schema}
Evidence (Hint): {evidence} (Optional)
Question: {question}

Output Format (JSON Only):
- Output must be a single JSON object with nodes and edges.
- Nodes: id (integer) and question (string).
- Edges: [parent_id, child_id], where parent depends on child.
- The graph must be a DAG, flowing from leaves to root.

Output only the valid JSON representation of the graph.

C.3 SQL Generator Prompt

SQL Generator Prompt - BIRD

System: You are an expert SQL generator. Below, you are presented with a database schema and a question. Your task is to read the schema, understand the question, and generate a valid {self.db_type} query to answer the question.

Input:
Database Schema: {schema_str}
Question: {sub_question}
Hint (Evidence): {evidence}

MANDATORY BIRD EXECUTION RULES:
- **1. The Hint is GROUND TRUTH:** The Hint (Evidence) is a strict requirement. If the hint provides a mapping (e.g., "'A"' for "'Active"') or a specific filter value, you MUST use it exactly as specified. Do not deviate from the hint.
- **2. Case-Insensitivity:** SQLite is case-sensitive for string comparisons. Use LOWER() on columns and filter values (e.g., LOWER(city) = LOWER('London')) or LIKE for all string filters unless the hint specifies otherwise.
- **3. Joins:** Use explicit JOIN ... ON syntax.
- **4. Name vs ID:** Always prefer using the name column over the id column unless the hint specifies otherwise.
- **5. Null Handling:** Be mindful of NULL values in your logic, especially in aggregations and comparisons.
- **6. Final Check:** Ensure your query returns all requested information and nothing extra.

Output: Please respond with a JSON object structured as follows:
{"SQL": "Your SQL query in a single string."}

Take a deep breath and think step by step to find the correct SQL query.

SQL Generator Prompt - Spider

System: You are a data science expert. Below, you are presented with a database schema and a question. Your task is to read the schema, understand the question, and generate a valid {self.db_type} query to answer the question. Before generating the final SQL query think step by step on how to write the query.

Input:
Database Schema: {schema_str}
Question: {sub_question}
Hint: {evidence}

This schema offers an in-depth description of the database's architecture, detailing tables, columns, primary keys, foreign keys, and any pertinent information regarding relationships or constraints. Special attention should be given to the examples listed beside each column, as they directly hint at which columns are relevant to our query.

Database instructions:
- Make sure you only output the information that is asked in the question. If the question asks for a specific column, make sure to only include that column in the SELECT clause, nothing more.
- Predicted query should return all of the information asked in the question without any missing or extra information.

Priority should be given to columns that have been explicitly matched with examples relevant to the question's context.

Output: Please respond with a JSON object structured as follows:
{
"chain_of_thought_reasoning": "Your thought process...",
"SQL": "Your SQL query in a single string."
}

Take a deep breath and think step by step to find the correct {self.db_type} SQL query.

C.4 SQL Refiner Prompt

SQL Refiner Prompt

System: You are an expert SQL debugger. Fix a failing SQL query using the full history of prior attempts and their execution errors.

Input:
Question: [QUESTION]
Database Schema: [SCHEMA]
Evidence: [EVIDENCE]
Attempt history ([N] failed attempts): [HISTORY]

Task: Review all prior failures and produce a corrected SQL query that addresses every identified issue. Do not include a trailing semicolon.

Output: Corrected SQL query.

C.5 Topology Refiner Prompt

Topology Refiner Prompt

System: You are an expert workflow planner. Your task is to revise a task decomposition graph that has failed during execution.

Input:
Original User Question: {question}
Current Decomposition Graph (JSON): {graph_json_str}
Execution Failure:
- The plan failed at **Node {failed_node_id}**.
- Sub-question for this node: "{sub-question}"
- Reason for failure: "{failure_reason}"

Your Task: Analyze the failure and the overall graph, then propose a corrected version of the graph in JSON format.

Revision Rules:
- **1. Localize Changes:** Make the minimum necessary changes to fix the issue. Focus on the area around the failed node.
- **2. Allowed Operations:**
  - Rephrase a node: Modify the 'question' of a node to be clearer or more specific.
  - Decompose a node: Replace a single failed node with two or more simpler nodes. Ensure you correctly wire the new dependencies.
  - Add a node: Add a new intermediate step if a prerequisite was missing.
  - Rewire edges: Change the dependencies (edges) if the data flow was incorrect.
- **3. Preserve Completed Nodes:** Do NOT modify nodes that have already executed successfully (i.e., nodes that are dependencies of the failed node).
- **4. Maintain Graph Integrity:** The output must be a valid Directed Acyclic Graph (DAG). Do not introduce cycles.
- **5. Output Format:** Your output MUST be only the complete, revised graph in the same JSON format as the input. Do not include any other text, explanations, or markdown formatting.

Output: Revised Graph (JSON only):

```
{
"nodes": [...],
"edges": [...]
}
```

D Case Studies

We present two worked examples from the BIRD development set illustrating how DecoSearch decomposes complex questions into a dependency DAG and solves each sub-question in topological order. Both examples were solved correctly by DecoSearch while generating a direct SQL attempt failed to produce the right answer.

D.1 Example 1: Sequential Dependency Chain

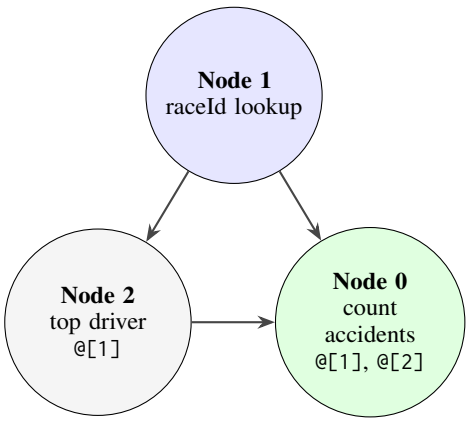
Question & Context — `formula_1` database

NLQ: How many accidents did the driver who had the highest number of accidents in the Canadian Grand Prix have?

Evidence: accidents ≡ rows where statusId = 3; Canadian Grand Prix ≡ races.name = 'Canadian Grand Prix'

Judger decision: Decompose. The question requires a two-phase aggregation: first materialise the set of Canadian GP race IDs, then identify the most-accident driver in those races, and finally count their accidents. Expressing this as a single MAX(COUNT(. . .)) subquery is error-prone in one-shot generation.

DAG (3 nodes; execution order $1 \rightarrow 2 \rightarrow 0$):



leaf *intermediate* *root*

Node	Deps.	Sub-question
1	—	Find the raceId(s) for the race named *'Canadian Grand Prix'*.
2	@[1]	Find the driverId with the most accidents (statusId=3) among races @[1].
0	@[1], @[2]	Count accidents for driver @[2] across races @[1].

Node 1 (leaf) — no dependencies

```
SELECT raceId FROM races
WHERE name = 'Canadian Grand Prix'
```

Node 2 — uses @[1]

```
SELECT driverId FROM results
WHERE statusId = 3 AND raceId IN @[1]
GROUP BY driverId ORDER BY COUNT(*) DESC LIMIT
1
```

Node 0 (root) — uses @[1], @[2]

```
SELECT COUNT(*) FROM results
WHERE driverId = @[2] AND statusId = 3 AND
raceId IN @[1]
```

Answer: 2 ✓ Matches gold.

Why decomposition helps: Direct queries for this pattern demand a nested `MAX(COUNT(...))` structure, a complexity that often causes LLMs to hallucinate or mistake it for a flat aggregation. By materializing the driver ID as an intermediate result, the task is simplified into basic `GROUP BY` operations or filters, which the model processes reliably.

D.2 Example 2: Parallel Branches with Aggregation

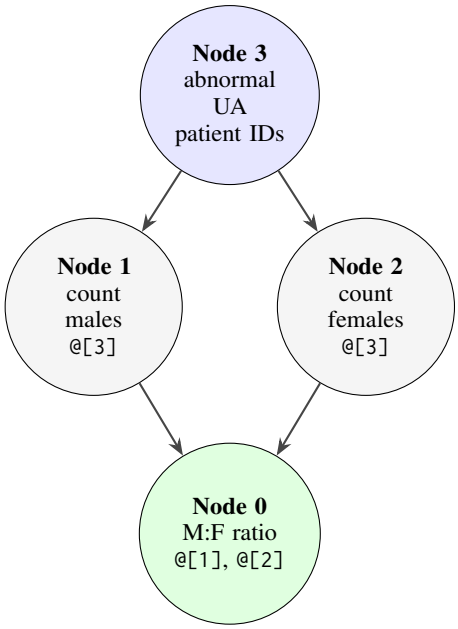
Question & Context — `thrombosis_prediction` database

NLQ: What is the ratio of male to female patients among all those with abnormal uric acid counts?

Evidence: male ≡ SEX='M'; female ≡ SEX='F'; abnormal UA ≡ UA≤8.0 for males, UA≤6.5 for females; ratio = DIVIDE(count_M, count_F)

Judger decision: Decompose. The ratio depends on two separate conditional counts with sex-specific thresholds. Decomposition computes each count independently, then combines them in the root node, avoiding the compound CASE-within-CASE logic that often produces off-by-one errors in direct generation.

DAG (4 nodes; execution order $3 \rightarrow 1, 2 \rightarrow 0$; diamond topology):



leaf *intermediate* *root*

Node	Deps.	Sub-question
3	—	Find all patient IDs where UA is abnormal: (SEX='M' & UA≤8.0) OR (SEX='F' & UA≤6.5).
1	@[3]	Count distinct male patients (SEX='M') from @[3].
2	@[3]	Count distinct female patients (SEX='F') from @[3].
0	@[1], @[2]	Compute CAST(@[1] AS REAL) / @[2].

Node 3 (leaf) — no dependencies

```
SELECT DISTINCT l.ID FROM Laboratory l
JOIN Patient p ON l.ID = p.ID
WHERE (p.SEX = 'M' AND l.UA <= 8.0)
      OR (p.SEX = 'F' AND l.UA <= 6.5)
```

Node 1 — uses @[3]

```
SELECT COUNT(DISTINCT p.ID) FROM Patient p
WHERE p.ID IN @[3] AND p.SEX = 'M'
```

Node 2 — uses @[3]

```
SELECT COUNT(DISTINCT p.ID) FROM Patient p
WHERE p.ID IN @[3] AND p.SEX = 'F'
```

Node 0 (root) — uses @[1], @[2]

```
SELECT CAST(@[1] AS REAL) / NULLIF(@[2], 0)
AS ratio
```

Answer: 0.2057 ✓ Matches gold.

Why decomposition helps: The sex-specific abnormality thresholds make it natural to separate the population filtering (Node 3) from the sex-conditional counts (Nodes 1 and 2). A direct query

must interleave both concerns in a single CASE expression, a pattern where LLMs frequently apply the wrong threshold to the wrong gender. The diamond topology here shows that DecoSearch is not limited to linear chains: two sibling nodes can share a common dependency and produce independent partial results that are combined at the root.

E LLM Usage

In this paper, we leverage LLMs, including ChatGPT and Gemini, to refine sentence-level writing.