VET: Verifiable Execution Tracing for Reliable Text-to-SQL Generation

Dongyu Wang[1] dywang3254@mail.ustc.edu.cn
Jingyu Li[1,*] jingyuli@iai.ustc.edu.cn
Lan Zhang[1] zhanglan@ustc.edu.cn
Ganggang Yu[2] 13916942837@139.com
Liang Huang[2] exceedhl@gmail.com
[1]University of Science and Technology of China
[2]NIO

Abstract

Large language models (LLMs) have shown remarkable capabilities in text-to-SQL generation, yet existing approaches remain prone to hallucinations and lack verification mechanisms. Current methods such as Chain-of-Thought (CoT) and Program-of-Thought (PoT) typically rely on intermediate reasoning that is either purely textual or executed only as a final step, leaving the reasoning process opaque and prone to grounding and logical hallucinations. In this paper, we introduce Verifiable Execution Tracing (VET), a novel reasoning paradigm that transforms text-to-SQL from unverifiable textual rationales into step-wise executable semantics. VET addresses these limitations by constraining the reasoning process within a candidate schema space and formulating it as a sequence of executable Python steps. Crucially, each step is executed against the real database to produce observable intermediate results, which serve as immediate verification feedback and transform the traditionally opaque generation process into a transparent, debuggable interaction with database reality. Experiments show consistent gains under matched, training-free settings, achieving 70.93% execution accuracy on BIRD and 37.04% on Spider 2.0-lite, with particularly strong improvements on complex queries.

1 Introduction

Large language models (LLMs) have emerged as a transformative interface for translating natural language into SQL queries, enabling intuitive database interactions across diverse domains. However, despite their remarkable progress, generating reliable and semantically correct SQL queries remains a fundamental challenge, as benchmarks evolve towards rigorous industrial standards (Li et al., 2024b). The inherent ambiguity of natural language, coupled with the structural complexity of database schemas, creates a gap between user intent and executable queries that current approaches struggle to bridge effectively.

Current text-to-SQL systems face a primary limitation in reliably interpreting user intent within complex database environments. Users frequently provide incomplete or colloquial natural language descriptions, which leads to misinterpretations of field semantics and hallucinated relationships between entities. Recent studies have formalized these failures into a taxonomy of schema-based, logic-based, and content-based hallucinations (Yang et al., 2025; Qu et al., 2024). Even with sophisticated schema linking mechanisms designed to disambiguate retrieval and schema grounding (Wang et al., 2025c), models often struggle to establish accurate correspondences between vague user intents and specific database structures. This misalignment is particularly prevalent in in-context learning settings, where schema errors remain a dominant failure mode (Li et al., 2025b), resulting in queries that are syntactically valid but semantically incorrect.

To address these interpretive challenges, the field has increasingly adopted Chain-of-Thought (CoT) reasoning and its decomposed variants to explicitly structure the generation process. However, relying on these textual reasoning traces introduces a second, fundamental bottleneck: *unverifiability*. As depicted in the left panel of Figure 1, text-based reasoning approaches operate in a purely linguistic space, relying on implicit textual traces where the model essentially "guesses" the mapping between ambiguous intent and schema elements. This detachment from the execution environment makes models susceptible to "faithfully" executing initial errorssuch as hallucinating data values or violating schema constraintswithout detection, as the intermediate reasoning cannot be externally validated (Gao et al., 2023a). While some Program-of-Thought (PoT)

* Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 30867–30883

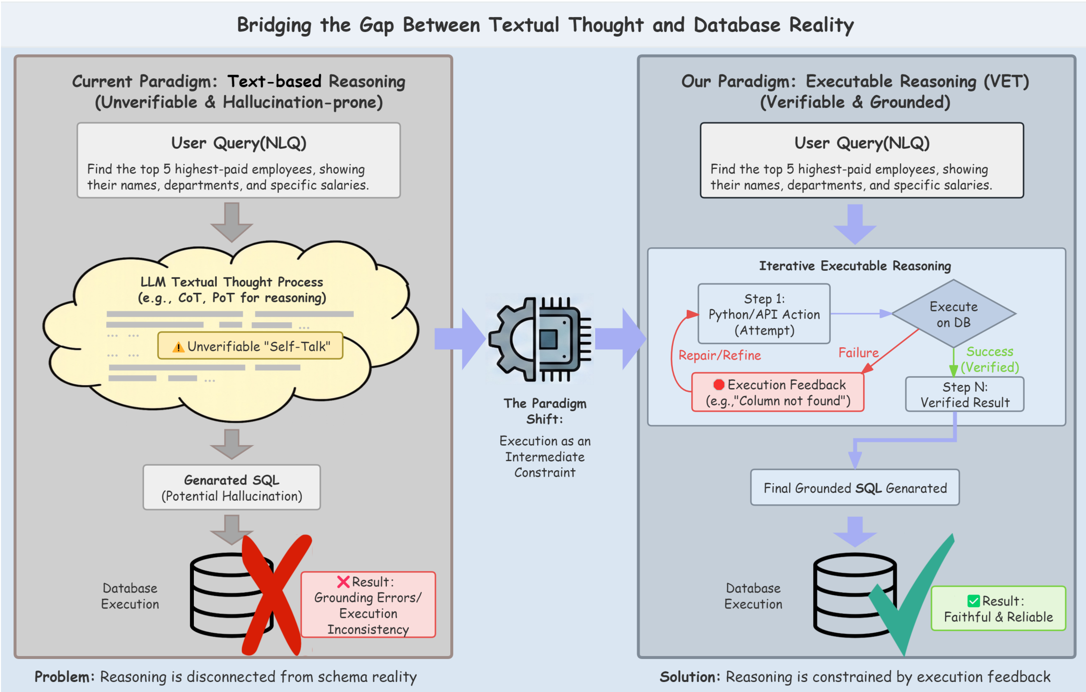

Figure 1: Comparison of reasoning paradigms. **Left:** Traditional text-based approaches operate as a black box, relying on ephemeral and ungrounded textual thoughts that are prone to hallucinations. **Right:** The VET framework utilizes executable code as a tangible carrier of reasoning. By materializing abstract intent into verifiable stepssuch as probing data existence in temporary DataFramesit transforms the process into a visible, debuggable interaction, ensuring logical consistency prior to SQL synthesis.

approaches introduce code execution to mitigate this (Gao et al., 2023c), they typically employ a "one-shot" execution paradigm where programs are validated only at the final stage. This delayed verification leaves intermediate logical errorssuch as redundant filters or incorrect aggregationsundetected until they manifest as incorrect final results. Consequently, the reasoning process remains opaque, lacking the fine-grained, step-by-step interpretability required to verify and correct decision-making traces (Nguyen et al., 2025; Liu et al., 2025).

To address the aforementioned limitations and leverage this execution-based paradigm, we propose Verifiable Execution Tracing (VET), a novel framework that transforms text-to-SQL from an unverifiable generation task into a step-wise executable reasoning process. Unlike emerging multi-agent frameworks that rely on inter-agent textual critique (Heidari et al., 2025; Wang et al., 2025a), VET models the generation process as a function of a validated execution trace: $Y = G(V(E(Q, S)))$. Contrasting with the black-box failure modes shown on the left, the right panel of Figure 1 highlights the core innovation of our approach: using executable code as the tangible carrier of reasoning. Instead of unverifiable textual thoughts, code serves as a rigorous medium that materializes abstract intent into verifiable logic steps. By executing these intermediate code stepssuch as verifying data existence in a temporary DataFrameVET transforms the reasoning process into a visible, debuggable interaction with the database, ensuring logic consistency before the final SQL synthesis.

The key contributions of our work can be summarized as follows:

- We propose VET (Verifiable Execution Tracing), a novel reasoning paradigm that transforms text-to-SQL from unverifiable generation to step-wise executable verification. By ensuring that every intermediate reasoning step is grounded in database reality, we eliminate the unverifiability inherent in traditional textual rationales.

- We employ executable code as the primary vehicle for reasoning, effectively resolving the semantic ambiguity of natural language through the precision of programming logic. Unlike textual CoT which permits vague logical leaps, code reasoning compels the model to materialize abstract intent into precise, executable operations, thereby mitigating hallucinations at the source.

- Under matched, training-free settings, VET delivers consistent gains on BIRD and Spider 2.0-lite, with the largest improvements on challenging queries, where the benefits of step-wise executable verification are most evident.

2 Related Work

Evolution of Text-to-SQL. The transformation of natural language into SQL has evolved from pattern matching to semantic parsing. Traditional approaches like RAT-SQL (Wang et al., 2019) and SMBOP (Rubin and Berant, 2020) utilized encoder-decoder models with relation-aware self-attention to align schema elements. With the rise of LLMs, the focus shifted to in-context learning. Methods such as DIN-SQL (Pourreza and Rafiei, 2024) and DAIL-SQL (Gao et al., 2023b) demonstrated that decomposing queries and selecting diverse examples significantly improves performance. However, these methods primarily operate in the textual domain. Recent advancements such as READ-SQL (Gao et al., 2023a) and PARSQL (Dai et al., 2025) have attempted to structure this reasoning process by decomposing queries into sub-components or utilizing parse trees to guide generation. While these decomposition strategies improve logical coherence, they often lack immediate grounding in the actual database state, leaving the "sim-to-real" gap unbridged.

Agentic and Execution-Guided Reasoning. Recent work has shifted towards execution-guided and agentic paradigms. Execution-guided methods such as ExCoT-DPO (Zhai et al., 2025) and Alpha-SQL (Li et al., 2025a) leverage execution feedback or search strategies (e.g., MCTS) to improve SQL generation. Agentic approaches like AgentiQL (Heidari et al., 2025) and MAC-SQL (Wang et al., 2025a) employ multi-agent architectures for planning, coding, and refinement. Unlike these prior methods, VET uniquely performs *inference-time semantic state search.* Python execution is used not only to validate the final output but also to probe intermediate states, effectively mitigating content-based hallucinations that prior methods struggle to detect.

Algorithm 1: Verifiable Execution Tracing

```
Input: Question Q, schema S, database D
Output: Executable SQL query Y
Schema-Constrained Initialization:
Extract candidate column set C* from (Q, S);
Initialize data state I_0 ← D;
Initialize reasoning trace P ← [];
Iterative Executable Reasoning:
while query intent not yet satisfied do
    Generate executable operation op_t constrained to C*;
    Execute op_t on I_{t-1} to obtain output O_t;
    if execution error or verification violation then
        Repair op_t using execution feedback;
        continue
    Append p_t = (op_t, I_{t-1}, O_t) to P;
    Update data state I_t ← O_t;
SQL Synthesis and Verification:
Generate SQL query Y conditioned on verified trace P;
Execute Y on D to obtain R_SQL;
if R_SQL ≠ O_k or Sim(Q, Q') < τ then
    Reject Y;
return Y
```

Execution-based Verification. Ensuring the correctness of generated SQL is critical. Approaches like R^3 (Xia et al., 2024) assess reliability by executing generated queries and checking for runtime errors. LEVER (Ni et al., 2023) takes this further by training a discriminator to rank SQL candidates based on their execution results. While effective, LEVER acts as a "Discriminative Black-box" relying on statistical likelihoods. VET introduces a "Generative White-box" approach via its Bidirectional Consistency Protocol, verifying validity by semantically reconstructing the user intent from the execution result, offering superior interpretability.

3 Methodology

3.1 Problem Definition

Let Q denote a natural language question, $\mathcal{S}$ a database schema consisting of tables and typed columns, and $\mathcal{D}$ the corresponding database instance. The goal of text-to-SQL is to generate an

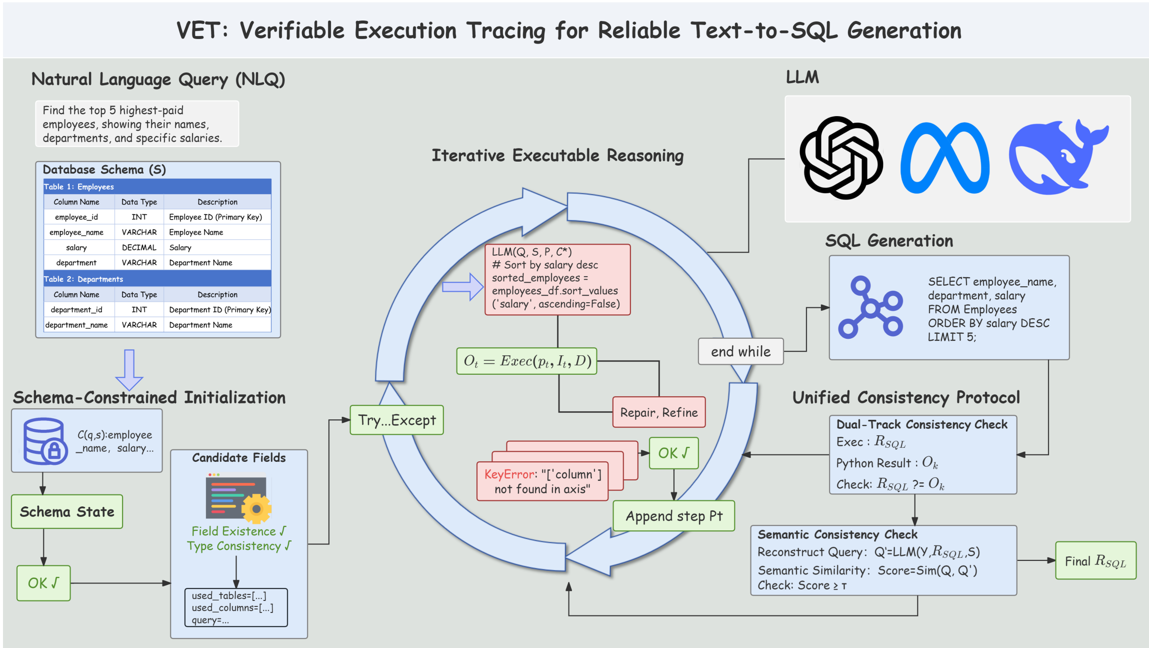

Figure 2: Overview of the proposed Verifiable Execution Tracing (VET) framework. VET first performs schema-constrained initialization to restrict the reasoning space to schema-grounded elements. It then constructs an iterative executable reasoning trace, where each generated operation is immediately executed on the database to produce observable feedback, which is used to verify, refine, or repair subsequent reasoning steps, with data probing operations invoked when ambiguity arises. Finally, VET synthesizes the SQL query from the verified execution trace and applies a unified consistency protocol, enforcing both execution alignment between SQL and reasoning outcomes and semantic consistency with the original question.

executable SQL query Y such that executing Y on $\mathcal{D}$ yields results that faithfully answer Q.

Most existing approaches model this task as a direct mapping from Q to Y, optionally augmented with textual intermediate reasoning. However, such formulations implicitly assume that intermediate reasoning steps are correct and grounded, despite lacking any mechanism to verify them against the actual database. As a result, errors such as hallucinated columns, invalid values, or logically inconsistent constraints often remain undetected until final execution. To address this limitation, we reformulate text-to-SQL as a structured reasoning problem with explicit execution-based verification. Instead of treating SQL generation as a one-shot output, we decompose it into reasoning, verification, and synthesis stages, enabling intermediate decisions to be validated before they affect the final query.

3.2 Overview of Verifiable Execution Tracing

We propose **Verifiable Execution Tracing (VET)**, a reasoning framework that transforms text-to-SQL from an unverifiable generation task into a step-wise executable verification process. VET represents reasoning not as free-form text, but as a sequence of executable operations whose effects can be directly observed on the database. The verified trace also stores compact output summaries and provenance metadata for later SQL reconstruction and optional auditing. Formally, VET models SQL generation as a composite function:

$$Y = \mathcal{G}\big(\mathcal{V}(\mathcal{E}(Q, \mathcal{S}), \mathcal{D})\big), \tag{1}$$

where $\mathcal{E}$ generates an executable reasoning trace constrained by the schema, $\mathcal{V}$ performs execution-based verification against the database, and $\mathcal{G}$ synthesizes the final SQL query conditioned on the verified trace.

As illustrated in Figure 2, VET consists of three tightly coupled phases: (1) Schema-Constrained Initialization, (2) Iterative Executable Reasoning with Data Probing, and (3) SQL Synthesis with a Unified Consistency Protocol.

3.3 Schema-Constrained Initialization

A fundamental failure mode of text-to-SQL systems is schema hallucination, where models reference tables or columns that do not exist in the database but appear semantically plausible. To mitigate this issue, VET begins by explicitly constructing a constrained reasoning space over the schema.

Given schema $\mathcal{S}$ and question Q, a schema linking module extracts a high-recall candidate column set $\mathcal{C}^*$ intended to cover the columns potentially relevant to the query. We implement schema linking in two stages. Stage 1 performs high-recall retrieval over tables and columns using lexical overlap and semantic similarity between question spans and schema names/descriptions, producing a compact candidate pool. Stage 2 resolves ambiguities through lightweight execution-grounded probes over this pool, such as value existence checks, categorical enumerations, and join-anchor validation, before the main reasoning loop begins. The resulting set $\mathcal{C}^*$ is then treated as a hard constraint: all subsequent reasoning operations must operate exclusively over $\mathcal{C}^*$.

By externalizing schema constraints as explicit variables, VET enables explicit detection of invalid reasoning steps. If a generated operation references a column outside $\mathcal{C}^*$, it is identified as a hallucinated schema element and rejected before execution. We further require provenance consistency: each later filter value, join path, or aggregation key must be traceable to the original question, the retained schema slice, or an observed execution result. Repair proposals that introduce unsupported columns, values, or joins are rejected, even when they are syntactically executable.

3.4 Iterative Executable Reasoning

At the core of VET is the construction of an executable reasoning trace $P = [p_1, p_2, \ldots, p_k]$, where each step p_t is defined as a triplet:

$$p_t = (op_t, I_t, O_t), \tag{2}$$

with op_t denoting an executable Python operation (e.g., filtering, joining, aggregation), I_t the input data state, and O_t the observable output obtained by executing op_t on $\mathcal{D}$. Each operation is required to be type-consistent and executable, ensuring that O_t is deterministically derived from I_t.

Unlike purely textual reasoning, each reasoning step in VET is executed immediately after generation. Execution failures, empty results, or type errors serve as explicit feedback signals that guide refinement of subsequent operations. If an execution error occurs (e.g., due to a non-existent column or invalid value), the system triggers a repair process that leverages the error message to refine the operation, thereby suppressing hallucinations at their source.

Data Probing for Ambiguity Resolution. Natural language questions frequently introduce ambiguity in value expressions and schema mappings. For instance, a categorical filter such as "New York" may correspond to multiple surface forms (e.g., "NY" or "NYC") in the database, or an entity name such as "Apple" may appear in multiple columns (e.g., `Brand` vs. `Category`). To resolve such ambiguity, VET incorporates a data probing mechanism that inspects database contents before committing to filtering constraints.

When ambiguity is detected, the model generates a probing operation (e.g., `df[col].unique()` or value existence checks) and executes it on $\mathcal{D}$. The resulting observations are used to refine the original operation, ensuring alignment between user intent and actual database values. This online probing strategy directly mitigates content-based hallucinations, where models generate values that are linguistically plausible but factually absent from the database instance.

Repair Budget and Stopping Criterion. After execution, each step is subjected to lightweight verification conditions that enforce basic logical soundness. For example, filtering operations are expected to satisfy $|O_t| \leq |I_t|$, while aggregation operations should reduce data dimensionality. If a step violates these conditions, it is repaired before being appended to the reasoning trace. Each step is allowed at most 3 repair attempts. After a valid step is appended, a separate zero-temperature intent-sufficiency prompt receives (Q, P, O_t) and must output a strict `YES` or `NO` decision only. A `YES` means the current execution state already contains sufficient evidence to reconstruct the final SQL; otherwise VET continues to the next step. In our implementation, the reasoning horizon is capped at 5 steps, after which SQL reconstruction proceeds from the latest verified trace.

3.5 Unified Consistency Protocol

Once a verified reasoning trace is obtained, VET synthesizes the final SQL query Y by conditioning on P. However, generating an executable SQL statement alone does not guarantee semantic correctness. We therefore reconstruct the final SQL under trace constraints and then apply two complementary consistency checks.

Trace-Conditioned SQL Reconstruction. The SQL synthesis prompt receives the original question, the reduced schema slice $\mathcal{C}^*$, and the verified trace P with operation summaries, key outputs, and provenance metadata. It is instructed to reconstruct exactly one SQL query that preserves the tables, predicates, join paths, grouping keys, and aggregations already validated in P, while forbidding new schema items or literals unsupported by the trace. This turns the last stage into evidence-constrained reconstruction rather than fresh free-form reasoning.

Dual-Track Execution Consistency. We execute the synthesized SQL Y on $\mathcal{D}$ to obtain result R_{SQL} and compare it against the final execution output O_k of the reasoning trace. The SQL is accepted only if $R_{SQL} = O_k$; otherwise, the generation is rejected. This dual-track consistency ensures alignment between symbolic SQL execution and executable reasoning semantics.

Semantic Consistency via Question Reconstruction. While execution alignment enforces computational correctness, it does not fully guarantee intent preservation. We therefore reconstruct a natural language question Q' from $(Y, R_{SQL}, \mathcal{S})$ and measure its semantic similarity to the original query Q. We embed both questions with `bge-m3`, compute cosine similarity, and use a fixed threshold $\tau = 0.7$. The SQL is accepted only if both execution consistency holds and $\text{Sim}(Q, Q') \geq \tau$, ensuring that the final output is both executable and semantically faithful.

Through this unified verification pipeline, VET ensures that the generated SQL is not only correct by execution, but also grounded in a transparent, verifiable reasoning trajectory aligned with user intent.

4 Experimental Setup

4.1 Datasets and Benchmarks

We evaluate VET on three benchmark groups that stress distinct reasoning capabilities:

- **BIRD-Dev** (Li et al., 2024b): A large-scale, cross-domain dataset emphasizing complex database values and long-chain reasoning. We report Execution Accuracy (EX) on the development set.
- **Spider Variants** (Yu et al., 2019): We employ the standard **Spider** dataset alongside its robustness variants, **Spider-DK** (Domain Knowledge) and **Spider-SYN** (Synonym), to evaluate the model's resilience against linguistic perturbations and domain-specific terminology.
- **Spider 2.0-lite** (Lei et al.): A newer and more realistic benchmark with stronger schema-linking demands, higher compositional complexity, and fewer shortcut patterns than earlier Spider-style evaluations.

4.2 Baselines

We compare VET against a comprehensive set of state-of-the-art methods spanning three representative paradigms.

Text-centric prompting methods include Zero-shot and Few-shot Chain-of-Thought prompting, as well as decomposition-based approaches such as DIN-SQL (Pourreza and Rafiei, 2024) and DAIL-SQL (Gao et al., 2023b). These methods rely exclusively on textual reasoning and prompt engineering without external execution feedback.

Program-aided generation methods leverage code as an intermediate reasoning representation. We include TA-SQL (Qu et al., 2024) and Pi-SQL (chi et al., 2025), which employ Python programs to decompose logical structures. Unlike VET, these methods adopt a static generation paradigm without iterative verification or state correction.

Agentic and retrieval-augmented frameworks represent the current state-of-the-art in leveraging external knowledge and reranking mechanisms, including MAC-SQL (Wang et al., 2025b), CHESS (Talaei et al., 2024), Super-SQL (Li et al., 2024a), and RSL-SQL (Cao et al., 2024).

4.3 Implementation Details

Our primary backbone model is **DeepSeek-Chat (V3)**. To demonstrate model agnosticism, we also conduct experiments using **Qwen2.5-Coder** (7B, 14B, 32B) and **Llama3.1-8B**. All generations use temperature 0. VET allows at most 3 repair attempts for each reasoning step and at most 5 reasoning steps in total. The stopping decision is made by a separate zero-temperature intent-sufficiency prompt that must return only `YES` or `NO`.

Table 1: Execution accuracy on BIRD-Dev for various backbone models and methods, broken down by query difficulty (Simple, Moderate, Challenging).

Backbone	Method	Simple	Moderate	Challenging	Overall
GPT-4	TA-SQL	63.14	48.60	36.11	56.19
	SuperSQL	66.90	46.50	43.80	58.50
	MAC-SQL	65.73	52.69	40.28	59.39
Qwen2.5-32B	Pi-SQL	70.92	56.47	49.66	64.54
DeepSeek-Chat	Baseline	56.97	38.49	29.17	48.76
	CoT	57.19	39.78	27.78	49.15
	DIN-SQL	54.38	44.30	31.25	49.15
	DAIL-SQL	59.89	44.52	35.42	52.93
	TA-SQL	61.95	45.38	39.58	54.82
	CHESS	63.68	48.17	42.36	56.98
	RSL-SQL	69.73	54.09	54.48	63.56
	VET (Ours)	**72.54**	**62.37**	**61.11**	**68.38**
GPT-4o	Baseline	64.65	48.60	40.28	57.50
	RSL-SQL	74.38	57.11	53.79	67.21
	VET (Ours)	**76.11**	**64.09**	**59.72**	**70.93**

Table 2: Qualitative context on BIRD-Dev against selected fine-tuned leaderboard models.

Model	Accuracy (%)			
	Simple	Moderate	Challenging	Overall
Agentar-Scale-SQL	79.35	69.40	64.14	74.90
XiYan-SQL	-	-	-	73.34
CHASE-SQL	-	-	-	73.01
Q-SQL	-	-	-	72.99
OmniSQL-32B	-	-	-	69.23
VET (Ours)	**76.11**	**64.09**	**59.72**	**70.93**

Final SQL is reconstructed from the verified trace with a constrained prompt that forbids introducing new tables, columns, joins, or literals unsupported by provenance in P. For semantic consistency, we encode Q and Q' with `bge-m3`, compute cosine similarity, and set the fixed threshold to $\tau = 0.7$.

5 Results and Analysis

We conduct a comprehensive evaluation of VET to validate its effectiveness across performance, generalization, robustness, and mechanistic dimensions. All experiments adhere to a reproducible setup with a fixed temperature (0) and a maximum of 3 repair iterations to eliminate stochastic variability in LLM outputs.

5.1 Performance on BIRD-Dev

Table 1 presents the main execution accuracy results on BIRD-Dev across difficulty levels. With DeepSeek-Chat as the backbone, VET achieves an overall EX of 68.38%, outperforming the strongest matched baseline RSL-SQL (63.56%) by 4.82 points. When paired with GPT-4o, VET reaches 70.93%, exceeding GPT-4o + RSL-SQL by 3.72 points on the same split.

Notably, the largest gains appear in the *Challenging* subset, where VET attains 61.11% accuracy, surpassing DIN-SQL by 29.86 points and CHESS by 18.75 points. This subset is dominated by queries involving nested aggregation, multi-stage filtering, and value-dependent conditions. Purely text-based decomposition methods can accumulate early-stage reasoning errors in such cases, whereas VET detects and corrects erroneous intermediate states through step-wise execution feedback before they propagate to the final SQL.

For broader context, the training-free GPT-4o + VET configuration remains competitive with several fine-tuned systems reported on the BIRD leaderboard. We treat this comparison as qualitative context only, since those systems may involve additional training, reranking, different data splits, or proprietary evaluation pipelines.

5.2 Generalization Across Model Scales

Table 3: Performance of VET across different backbone models and model sizes on BIRD-Dev.

Model	Method	Sim.	Mod.	Cha.	Avg.
Qwen2.5-7B	Baseline	45.19	29.25	21.53	38.14
	VET	**58.59**	**42.80**	**36.11**	**51.69**
Qwen2.5-14B	Baseline	52.97	30.11	34.72	44.33
	VET	**68.43**	**54.19**	**49.31**	**62.32**
Llama3.1-8B	Baseline	34.70	22.37	22.22	29.79
	VET	**59.46**	**41.94**	**39.58**	**52.28**

To evaluate generalization across model capacities, we apply VET to backbone models of varying sizes and families, with results shown in Table 3.

Across all backbones, VET consistently improves execution accuracy. Notably, the relative gains are larger for smaller models (e.g., 7B), suggesting that VET functions as an external reasoning scaffold that compensates for limited parametric capacity through structured execution feedback.

5.3 Robustness under Distribution Shifts

We evaluate robustness on Spider, Spider-DK, and Spider-SYN, with results summarized in Table 4. These datasets progressively introduce linguistic and domain-level perturbations while preserving the underlying database schemas.

On the standard Spider benchmark, VET improves overall execution accuracy from 74.7% to 84.9%, with the largest gain observed in the Hard

Table 4: Execution Accuracy (EX) on Spider, Spider-DK, and Spider-SYN under distribution shifts.

Dataset	Method	Easy	Medium	Hard	Overall
Spider	Baseline	92.7	82.3	64.9	74.7
	+VET	**94.8**	**90.1**	**78.7**	**84.9**
Spider-DK	Baseline	**89.1**	80.9	51.4	70.7
	+VET	83.6	**85.0**	**73.0**	**80.4**
Spider-SYN	Baseline	84.7	72.0	59.3	65.8
	+VET	**92.3**	**84.8**	**76.8**	**80.7**

subset (64.9% → 78.7%). This indicates that even in in-domain settings, executable step-wise verification substantially benefits complex logical queries.

Spider-DK evaluates robustness to domain-specific terminology that deviates from common lexical priors. VET achieves an overall accuracy of 80.4%, outperforming the baseline by 9.7 points. This improvement stems from VET's data probing mechanism, which validates the existence and semantics of domain-specific values directly against the database, reducing reliance on brittle language priors.

Spider-SYN focuses on synonym substitution at the schema level. VET improves overall accuracy by 14.9 points (65.8% → 80.7%). When an incorrect synonym leads to empty execution results, VET automatically explores alternative mappings through execution feedback, eliminating the need for manually curated synonym dictionaries.

5.4 Results on Spider 2.0-lite

Spider 2.0-lite targets more realistic and more compositionally difficult settings, and is therefore well aligned with our focus on robustness under complex queries. With DeepSeek-Chat as the backbone and under matched conditions—no task-specific fine-tuning, no RL, no external training data, and no ensembling—VET reaches 37.04, improving over the DeepSeek-Chat baseline (14.82) by 22.22 points and over DeepSeek-Chat + RSL-SQL (26.14) by 10.90 points. Cross-backbone numbers such as RSL-SQL/OpenAI o3 and ReFoRCE/Qwen3 provide only broader context rather than strict apples-to-apples comparisons.

The ablation block in Table 5 shows that the gain comes from the full verification-and-repair pipeline instead of a single component. Removing any module degrades performance, and removing execution feedback lowers the score from 37.04 to 34.82, indicating that intermediate execution signals remain useful even in this harder, more robustness-oriented setting.

Table 5: Main results and ablations on Spider 2.0-lite.

Method	Backbone	Score
DIN-SQL	GPT-4o	1.46
CHESS	GPT-4o	3.84
DAIL-SQL	GPT-4o	5.68
Baseline	DeepSeek-Chat	14.82
LinkAlign	DeepSeek-Chat	24.86
CoT	DeepSeek-Chat	25.93
RSL-SQL	DeepSeek-Chat	26.14
RSL-SQL	OpenAI o3	33.09
ReFoRCE	Qwen3	35.60
VET (Ours)	**DeepSeek-Chat**	**37.04**
Ablation on DeepSeek-Chat		
Baseline	DeepSeek-Chat	14.82
w/o UC	DeepSeek-Chat	30.37
w/o SC	DeepSeek-Chat	32.60
w/o SV	DeepSeek-Chat	32.60
w/o EF	DeepSeek-Chat	34.82
VET (Full)	**DeepSeek-Chat**	**37.04**

5.5 Ablation Study

Table 6: Ablation results of VET on BIRD-Dev. Each variant removes one component from the full executable reasoning pipeline while keeping the remaining modules unchanged.

Variant	Simple	Mod.	Cha.	Overall
VET (Full)	**72.54**	**62.37**	**61.11**	**68.38**
w/o Schema Constr.	72.22	58.71	52.78	66.30
w/o Step-wise Verif.	70.92	53.98	47.22	63.56
w/o Unified Consist.	70.59	52.04	52.78	63.30
w/o Execution Feed.	70.81	52.90	45.83	63.04

We conduct an ablation study on BIRD-Dev to analyze the contribution of each component in VET. Table 6 reports variants where one module is removed while the others remain unchanged. Removing SC yields a moderate overall drop (68.38% → 66.30%), showing that early schema grounding stabilizes intent interpretation. Removing SV or UC causes larger losses (to 63.56% and 63.30%), confirming that one-shot execution and unconstrained final reconstruction both weaken reliability. Removing EF produces the largest degradation, especially on challenging queries (61.11%

$\rightarrow$ 45.83%), highlighting the value of grounding repairs in real database feedback. The same qualitative pattern reappears on Spider 2.0-lite (Table 5), where every removal hurts and EF remains the most beneficial single component.

5.6 Qualitative Case Studies Overview

To further illustrate how VET transforms potential failures into successful queries, we provide detailed qualitative analyses in Appendix A. Table 10 in the appendix presents three representative scenarios from BIRD-Dev: Schema Hallucination, where baseline models make single-table assumptions that VET corrects via KeyError-driven table expansion; Value Mismatch, where string formatting errors are caught and corrected via execution feedback; and Semantic Ambiguity, where multi-hop schema paths are discovered through failed execution signals.

These examples highlight the importance of step-wise verification and in-process repair in reducing logic, schema, and value errors. Additionally, Appendix D presents a full end-to-end case study, demonstrating VET's reasoning process from the initial query to the final SQL output in complex multi-step scenarios.

5.7 Error Patterns and Repair Mechanism

To complement quantitative performance, we analyze representative error types and the effect of VET's repair loop. Detailed qualitative error analysis and repair statistics are provided in Appendix B. Overall, VET substantially reduces logic and schema errors and recovers a meaningful fraction of otherwise failing queries, highlighting the practical benefits of step-wise verification and repair.

5.8 Efficiency Considerations

VET introduces iterative execution overhead due to step-wise verification and repair. However, this overhead is modest relative to the gains in execution accuracy, especially for complex queries. Detailed efficiency statistics, including inference time and token consumption per query difficulty, are provided in Appendix C. Techniques like schema constraints and potential caching can further mitigate runtime in practice.

6 Conclusion

We presented VET, a training-free framework that mitigates ungrounded hallucinations in Text-to-SQL by enforcing an executable and verifiable reasoning process. By constraining the search space, leveraging data probing for ambiguity resolution, and applying unified consistency checks, VET aligns natural language intent with database reality. Experiments on BIRD, Spider variants, and Spider 2.0-lite show consistent gains under matched settings and across model families. Future work will focus on reducing inference latency and extending VET to settings with restricted database access.

7 Limitations

While VET significantly reduces hallucinations and improves accuracy, it introduces two primary limitations. First, the step-wise execution mechanism increases inference latency compared to direct text-to-SQL generation, as it requires waiting for database feedback at each reasoning step. Although this overhead is justified for high-stakes queries, it may be a bottleneck for real-time applications requiring millisecond-level latency. Second, deployment involves a privacy–observability trade-off. In a Transparent/Auditing mode, the system stores the verified trace, step outputs, and repair history for debugging and post-hoc inspection. In a Secure mode, only schema metadata, bounded execution summaries, and the final SQL are retained, while raw row-level outputs are suppressed, masked, or hashed. When direct execution is further restricted, VET can fall back to an execution-limited variant that permits only schema inspection and aggregate probes. This weakens ambiguity resolution but degrades gracefully because schema constraints and trace-conditioned reconstruction still restrict the hypothesis space.

8 Ethical Considerations

Execution-augmented text-to-SQL systems raise privacy and automation concerns. Since VET executes generated code against real databases, improper sandboxing may expose or modify data. We therefore recommend read-only credentials, bounded execution budgets, and explicit separation between the Secure and Transparent/Auditing modes described above. VET is intended to assist domain experts, not replace them.

Acknowledgments

This research was supported by the China National Natural Science Foundation (No. 62441228), the Science and Technology Tackling Program of Anhui Province (No. 202423k09020016), the China Postdoctoral Science Foundation (2025M771515), the Anhui Postdoctoral Scientific Research Program Foundation (2025C1166), and the USTC-NIO Smart Electric Vehicle Joint Lab.

References

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

Yongdong chi, Hanqing Wang, Zonghan Yang, Jian Yang, Xiao Yan, Yun Chen, and Guanhua Chen. 2025. Pi-sql: Enhancing text-to-sql with fine-grained guidance from pivot programming languages. *Preprint*, arXiv:2506.00912.

Yaxun Dai, Haiqin Yang, Mou Hao, and Pingfu Chao. 2025. PARSQL: Enhancing text-to-SQL through SQL parsing and reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 661–681, Vienna, Austria. Association for Computational Linguistics.

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

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

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023c. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Omid Reza Heidari, Siobhan Reid, and Yassine Yaakoubi. 2025. Agentiql: An agent-inspired multi-expert framework for text-to-sql generation. *Preprint*, arXiv:2510.10661.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql work.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to sql: Are we fully ready? *Proceedings of the VLDB Endowment*, 17(11):33183331.

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

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024b. 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.

Miaoran Li, Jiangning Chen, Minghua Xu, and Xiaolong Wang. 2025b. Hallucination detection in structured query generation via llm self-debating. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 16102–16113.

Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-sql. *Preprint*, arXiv:2502.11656.

Giang Nguyen, Ivan Brugere, Shubham Sharma, Sanjay Kariyappa, Anh Totti Nguyen, and Freddy Lecue. 2025. Interpretable llm-based table question answering. *Preprint*, arXiv:2412.12386.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

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.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. *arXiv preprint arXiv:2405.15307*.

Ohad Rubin and Jonathan Berant. 2020. Smbop: Semi-autoregressive bottom-up semantic parsing. *arXiv preprint arXiv:2010.12412*.

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*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a.

MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025b. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

Yihan Wang, Peiyu Liu, and Xin Yang. 2025c. Linkalign: Scalable schema linking for real-world large-scale multi-database text-to-sql. *Preprint*, arXiv:2503.18596.

Hanchen Xia, Feng Jiang, Naihao Deng, Cunxiang Wang, Guojiang Zhao, Rada Mihalcea, and Yue Zhang. 2024. r^3: "this is my sql, are you with me?" a consensus-based multi-agent system for text-to-sql tasks. *Preprint*, arXiv:2402.14851.

Bo Yang, Yinfen Xia, Weisong Sun, and Yang Liu. 2025. Hallucination detection for llm-based text-to-sql generation via two-stage metamorphic testing. *Preprint*, arXiv:2512.22250.

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. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. Excot: Optimizing reasoning for text-to-sql with execution feedback. *Preprint*, arXiv:2503.19988.

A Qualitative Case Studies

In this appendix, we provide a detailed qualitative analysis of the repair mechanisms in VET to demonstrate how execution feedback transforms potential failures into successful queries. Table 10 presents three representative scenarios selected from the BIRD development set, covering Schema Hallucination, Value Mismatch, and Semantic Ambiguity.

In the first scenario (**Schema Hallucination**), the baseline model falls victim to 'single-table bias' assuming all necessary columns exist in the primary table. VET detects this via a `KeyError` during execution, which forces the model to expand its search space to related tables (e.g., `frpm`) and generate the correct JOIN condition.

The second scenario (**Value Mismatch**) highlights the fragility of string matching. The baseline blindly assumes a spaced format (`' = '`), resulting in zero matches. VET captures this empty result set as a negative signal (`ValueError` on argmax), triggering a re-examination of the literal values and leading to the correct tight format (`'='`).

Finally, the third scenario (**Semantic Ambiguity**) addresses complex schema linking. The baseline conflates 'set translations' with 'foreign data', a common semantic drift. The execution failure of the direct merge path compels VET to discover the valid multi-hop path through the `cards` table.

These cases collectively illustrate that VET's interactive Python layer serves as a crucial grounding mechanism, allowing the model to 'fail fast' and self-correct before committing to a final SQL query.

B Error Patterns and Repair Efficacy

Table 7: Error distribution analysis on 100 sampled error cases. VET significantly reduces errors across all categories, particularly Logic and Schema errors.

Method	Logic Error	Schema Error	Value Error
Baseline (CoT)	66.2%	27.7%	6.1%
VET (Ours)	**45.9%**	**11.4%**	**2.9%**
Reduction (Δ)	$\downarrow$ 20.3%	$\downarrow$ 16.3%	$\downarrow$ 3.2%

We complement quantitative results with a qualitative analysis of 100 sampled error cases (Table 7) and repair loop effectiveness (Table 8). VET reduces logic errors by 20.3 pp (66.2% $\rightarrow$

Table 8: Effectiveness of the repair loop on BIRD-Dev. "Success Rate" is calculated based on the triggered queries ($N_{trigger} = 156$).

Metric	Count	Global Rate	Success Rate
Total Queries	1534	100%	-
Triggered ($N_{trigger}$)	156	10.17%	-
Code Fixed	135	8.80%	86.54%
Final Correct	**82**	**5.35%**	**52.56%**

45.9%)—the largest reduction across error types—driven by the Logical Observability check, which flags non-sensical intermediate states (e.g., empty filter results) and triggers repair before final SQL generation. Schema errors are cut by 16.3 pp (27.7% $\rightarrow$ 11.4%) due to SC and Data Probing, as the model verifies column/table existence before generating operations. Value errors are reduced by 3.2 pp (6.1% $\rightarrow$ 2.9%), with Data Probing resolving format mismatches (e.g., ' = ' vs. '=') by querying actual data distributions.

The repair loop is triggered for 10.17% of queries (156/1534), with an 86.54% code-fix rate (135/156) and a 52.56% final correction rate (82/156). This means VET recovers 5.35% of queries that would otherwise fail—a critical gain for high-stakes applications like business intelligence. Table 10 highlights three representative failure modes addressed by VET: (1) schema hallucination (resolved via KeyError feedback to switch to the correct table), (2) value format mismatch (fixed via empty sequence error triggering repair), and (3) multi-hop schema path error (corrected via execution failure forcing identification of valid multi-hop paths). These cases illustrate VET's paradigm shift from post-hoc error correction to in-process validation, transforming unverifiable textual errors into actionable execution feedback.

C Efficiency Statistics

While VET introduces iterative execution overhead (Table 9), the trade-off between performance and efficiency is favorable. Inference time for VET is modestly higher than baselines (e.g., 56.5s for Challenging queries vs. 40.0s for the baseline without EF), but this is offset by a 15.28 pp accuracy gain in the same subset. Token consumption remains competitive (17k tokens for Challenging queries vs. 35k for baselines without SC), as schema constraints reduce the model's search space and limit redundant token generation. Compared to multi-agent methods like MAC-SQL, VET's deterministic verification loop avoids consensus-based sampling overhead, resulting in lower token usage despite iterative repair. For real-world applications, the marginal increase in inference time is justified by substantial accuracy gains and the 5.35% recovery of failed queries via repair loops.

Table 9: Comparison of average inference time (s) and token consumption. VET maintains reasonable efficiency while delivering superior accuracy.

Method	Simple		Moderate		Challenging	
	Time	Tok.	Time	Tok.	Time	Tok.
VET	42.4	14k	49.7	16k	56.5	17k
w/o Schema	50.5	33k	47.6	36k	90.5	35k
w/o Step.	41.2	12k	47.7	15k	116.8	15k
w/o Unified	40.3	13k	47.0	15k	52.7	14k
w/o Feed.	40.0	14k	47.8	16k	78.5	18k

Table 10: **Qualitative Analysis of Error Repair.** We present three representative cases where the Baseline fails due to schema hallucination, value mismatch, and logic errors. In contrast, VET utilizes the Python execution feedback (e.g., `KeyError`, `ValueError`) as a grounded signal to detect inconsistencies and iteratively refine the reasoning process, leading to correct SQL generation.

Query & SQL Comparison	Interactive Repair Process (Python)	Error Analysis

Q: What is the lowest grade for the District Special Education Consortia School (ID 613360)?
Baseline (✗):
`SELECT MIN(LowGrade) FROM schools ...`
→ Hallucinates 'LowGrade' in 'schools'.
VET (✓):
`SELECT MIN(T1.'Low Grade') FROM frpm AS T1 JOIN schools AS T2 ...`
	[Trigger] Execution Failure:
	`# Assume data is in 'schools' table`
	`low_grade = schools['Low Grade'].min()`
	Error: `KeyError: 'Low Grade'`
	(Probe: Column missing in 'schools')
	↓ *Self-Correction via Repair Loop*
	[Repair] Cross-Table Retrieval:
	`# Switch to Join 'frpm' table`
	`merged = pd.merge(schools, frpm, ...)`
	`result = merged['Low Grade'].min()`
	Result: `Execution Success` ✓
		Diagnosis:
		Single-table Bias. The Baseline fails to locate data in the external `frpm` table.
		
		Effect:
		Schema Probe. The `KeyError` invalidates the single-table assumption. This forces the model to switch strategies, retrieving the correct table and generating correct **JOIN** logic.

Q: Is the molecule with the most double bonds carcinogenic?
Baseline (✗):
`... WHERE bond_type = ' = ' ...`
→ Value Error: Extra spaces cause zero matches.
VET (✓):
`SELECT ... WHERE T1.bond_type = '=' ...`
	[Trigger] Execution Failure:
	`# Filter by string with spaces`
	`bonds = df[df['bond_type'] == ' = ']`
	`top_mol = bonds...idxmax()`
	Error: `ValueError: attempt to get argmax of an empty sequence`
	(Probe: Query returned 0 rows)
	↓ *Self-Correction via Repair Loop*
	[Repair] Robust Handling → Correct SQL:
	`# Code robustly handles empty case`
	`if not bonds.empty: ...`
	Result: `Execution Success` ✓
	Feedback: The "Empty" signal guided the final SQL to use the correct literal `'='`.
		Diagnosis:
		String Literal Mismatch. The Baseline incorrectly infers the data format as `' = '` (with spaces), leading to an empty result set.
		
		Effect:
		Execution Feedback. The `ValueError` (empty sequence) acts as a negative signal, indicating the string literal was incorrect. This feedback steers the model to generate the correct, tight-formatted string `'='` in the final SQL.

Q: List top 2 sets in 'Shadowmoor' block that have Italian-language cards.
Baseline (✗):
`... FROM sets JOIN set_translations ON ... WHERE language='Italian'`
→ Logic Error: Wrong table. Checks set name translation, not card content.
VET (✓):
`... FROM sets T1 JOIN cards T2 ... JOIN foreign_data T3 ... WHERE T3.language='Italian'`
	[Trigger] Execution Failure:
	`# Attempt direct link between Sets and ForeignData`
	`merged = pd.merge(sets, foreign_data, on='id')`
	Error: `KeyError: "['name'] not in index"`
	(Direct merge failed; ambiguous columns/no path)
	↓ *Self-Correction via Repair Loop*
	[Repair] Multi-hop Path Found:
	`# Recognized missing intermediate link`
	`# Path: Sets → Cards → ForeignData`
	Result: `Execution Success` ✓
		Diagnosis:
		Semantic Ambiguity. Baseline confused "translated set names" ('set_translations') with "foreign language cards" ('foreign_data').
		
		Effect:
		Path Validation. The execution failure invalidates the incorrect or direct join attempts. This constrains the model to search for a valid schema path, correctly identifying 'cards' as the necessary intermediate bridge.

D End-to-End Execution Case Study

In this section, we present a comprehensive end-to-end case study demonstrating the VET process on a sample query from the BIRD-Dev dataset. This example illustrates how the model initializes the environment, performs step-wise executable reasoning, and validates the final output.

D.1 Phase 1: Initialization & Schema Loading

The process begins with the initialization of the database schema and the loading of relevant tables.

Query: "What is the highest eligible free rate for K-12 students in the schools in Alameda County?"
Evidence: Eligible free rate for K-12 = 'Free Meal Count (K-12)' / 'Enrollment (K-12)'
Database: `california_schools`
Selected Tables: `schools`, `satscores`, `frpm`

The system first defines the schema structure and loads the data into Pandas DataFrames.

```
[Python]
import pandas as pd
import numpy as np
from numpy import nan
pd.options.mode.chained_assignment = None
#`schools` data
schools_data = [
    ('37679916037576', 'San Diego', 'Bostonia Language Academy', 'K-3'),
    ('15635291530369', 'Kern', 'Centennial High', '9-12'),
    ('41689990134197', 'San Mateo', 'Aspire East Palo Alto Charter', None)
]

#`schools` primary key: `CDSCode`
schools_columns = ['CDSCode', 'County', 'School', 'GSserved']

schools = pd.DataFrame(schools_data, columns=schools_columns)
#`satscores` data
satscores_data = [
    ('10623800000000', 'Fresno', 291),
    ('1612000133397', 'Alameda', 454),
    ('39686270130864', 'San Joaquin', 116)
]

#`satscores` primary key: `cds`
#`satscores` foreign key: ['cds']
satscores_columns = ['cds', 'cname', 'enroll12']

satscores = pd.DataFrame(satscores_data, columns=satscores_columns)
#`frpm` data
frpm_data = [
    ('19647330129858', 'Los Angeles', 170.0, 149.0, 0.876470588235294),
    ('09737836110498', 'El Dorado', 19.0, 19.0, 1.0),
    ('33670903330412', 'Riverside', 1608.0, 958.0, 0.595771144278607)
]

#`frpm` primary key: `CDSCode`
#`frpm` foreign key: ['CDSCode']
frpm_columns = ['CDSCode', 'County Name', 'Enrollment (K-12)', 'Free Meal Count (K-12)', 'Percent (%)
     Eligible Free (K-12)']

frpm = pd.DataFrame(frpm_data, columns=frpm_columns)
```

```
[Python]
import sqlite3

conn = sqlite3.connect("dev/dev_databases/california_schools/california_schools.sqlite")

query = "SELECT * FROM `satscores`;"
satscores = pd.read_sql_query(query, conn)
satscores = satscores[['cds', 'cname', 'enroll12']]

query = "SELECT * FROM `schools`;"
```

```
schools = pd.read_sql_query(query, conn)
schools = schools[['CDSCode', 'County', 'School', 'GSserved']]

query = "SELECT * FROM `frpm`;"
frpm = pd.read_sql_query(query, conn)
frpm = frpm[['CDSCode', 'County Name', 'Enrollment (K-12)', 'Free Meal Count (K-12)', 'Percent (%)
    Eligible Free (K-12)']]
```

```
[Python]
query = """ What is the highest eligible free rate for K-12 students in the schools in Alameda County
    ? """
evidence = """ Eligible free rate for K-12 = `Free Meal Count (K-12)` / `Enrollment (K-12)` """
```

D.2 Phase 2: Iterative Reasoning

The model then proceeds with a step-by-step reasoning process, executing Python code at each stage to verify intermediate results.

D.2.1 Step 1: Filter Data by County

```
[Markdown]
### Step 1: Filter target data for Alameda County

To answer the question "What is the highest eligible free rate for K-12 students in the schools in
    Alameda County", we first need to focus on relevant data for **Alameda County**:

- Core Dataset: The `frpm` table contains key metrics such as K-12 enrollment and free meal counts
    for each county.
- Filtering Logic: Filter rows where the `County Name` field equals `'Alameda'` to obtain a dataset
    `alameda_frpm` containing only Alameda County, narrowing the scope for subsequent ratio
    calculations.
```

```
[Python]
# Step 1: Filter data for Alameda County
alameda_frpm = frpm[frpm['County Name'] == 'Alameda']

# Print filtered result
print(alameda_frpm)
```

```
[Execution Output]
   CDSCode County Name  Enrollment (K-12)  Free Meal Count (K-12)  \
0  01100170109835       Alameda           1087.0                    565.0
1  01100170112607       Alameda            395.0                    186.0
2  01100170118489       Alameda            244.0                    134.0
3  01100170123968       Alameda            191.0                    113.0
4  01100170124172       Alameda            257.0                     14.0
..            ...           ...              ...                      ...
374  01751016108971       Alameda           1258.0                     98.0
375  01751016114318       Alameda            662.0                     25.0
376  01751016117592       Alameda           1164.0                     45.0
377  01751016117600       Alameda            723.0                     33.0
378  01751196002711       Alameda            278.0                     14.0

      Percent (%) Eligible Free (K-12)
0                         0.519779
1                         0.470886
2                         0.549180
3                         0.591623
4                         0.054475
..                             ...
374                       0.077901
375                       0.037764
376                       0.038660
377                       0.045643
378                       0.050360

[379 rows x 5 columns]
```

D.2.2 Step 2: Calculate Eligible Free Rate

```
[Markdown]
### Step 2: Calculate eligible free rate per school

Based on the filtered `alameda_frpm` dataset, the next step is to calculate the eligible free meal
    rate for K-12 students in each school:

- Calculation Basis (Core Formula):
  > Eligible free rate for K-12 = `Free Meal Count (K-12)` / `Enrollment (K-12)`

- Operational Logic: Add a new column `Eligible Free Rate` to `alameda_frpm` to store the free meal
    eligibility rate for each row (individual school). This is the core calculation step for finding
     the "highest rate".
```

```
[Python]
# Step 2: Calculate the eligible free rate for each school
alameda_frpm['Eligible Free Rate'] = alameda_frpm['Free Meal Count (K-12)'] / alameda_frpm['
    Enrollment (K-12)']

# Print calculation result
print(alameda_frpm)
```

```
[Execution Output]
   CDSCode County Name  Enrollment (K-12)  Free Meal Count (K-12)  \
0  01100170109835       Alameda           1087.0                  565.0
1  01100170112607       Alameda            395.0                  186.0
...
378  01751196002711       Alameda            278.0                   14.0

      Percent (%) Eligible Free (K-12)  Eligible Free Rate
0                            0.519779           0.519779
1                            0.470886           0.470886
...
378                          0.050360           0.050360

[379 rows x 6 columns]
```

D.2.3 Step 3: Extract Maximum Rate

```
[Markdown]
### Step 3: Extract highest eligible free rate

After calculating the rate for individual schools, we need to extract the maximum value from the `
    Eligible Free Rate` column in `alameda_frpm`:

- Core Goal: Find the **highest value** of eligible free meal rates for K-12 students among all
    schools in Alameda County.
- Operational Logic: Use the Pandas `max()` method to directly obtain the maximum value of this
    column and assign the result to the variable `highest_eligible_free_rate`, precisely addressing
    the core requirement of the original question.
```

```
[Python]
# Step 3: Find the maximum eligible free rate
highest_eligible_free_rate = alameda_frpm['Eligible Free Rate'].max()

# Print highest rate
print(highest_eligible_free_rate)
```

```
[Execution Output]
1.0
```

D.3 Phase 3: Unified Consistency Protocol

Based on the verified reasoning trace, the system generates the final SQL query and performs consistency checks.

```
[SQL]
SELECT
  MAX(
    CAST(`Free Meal Count (K-12)` AS REAL) / `Enrollment (K-12)`
  ) AS highest_eligible_free_rate
FROM frpm
WHERE
  `County Name` = 'Alameda';
```

```
[Execution Output]
   highest_eligible_free_rate
0                         1.0

[1 rows x 1 columns]
```

Dual-Track Consistency Check: The execution result of the generated SQL ('1.0') is compared with the output of the final Python reasoning step ('highest_eligible_free_rate = 1.0'). Since the two values match, the SQL execution is consistent with the Python-based intermediate computation, confirming result reliability.

Semantic Consistency Check: We reconstruct the natural language query from the generated SQL and schema:

```
[Query]
What is the highest eligible free rate for K-12 students in the schools in Alameda County?
```

A cosine similarity calculation between the reconstructed query and the original question yields: Cosine Score = 0.8926 > τ, confirming semantic consistency.

Since both consistency checks pass, the final SQL is validated.