DATABOX: TASK-SPECIFIC SANDBOXES FOR EFFICIENT EXECUTION-GUIDED SQL DEBUGGING

Anonymous authors
Paper under double-blind review

ABSTRACT

The explosion of enterprise data and database-intensive applications has increased the complexity of SQL-based analytical and operational logic, raising the demand for automated query diagnosis and repair. Recent progress on SQL-based tasks has been driven by LLM-based agentic methods, which rely on execution-guided interaction loops with real database environments to iteratively inspect, reflect on, and refine candidate solutions. However, these agentic approaches typically require direct interaction with real databases, incurring high execution cost and latency on enterprise-scale applications and becoming impractical when privacy or deployment constraints restrict access to the underlying real data. To address these limitations, we propose Databox, an automatically constructed, task-specific sandbox environment that serves as a drop-in replacement for real-database interaction. Databox combines task-specific schema extraction, diagnostic value generation, and consistency validation, enabling private, low-latency, and compute-efficient execution-guided SQL debugging without accessing the original database records. To further reduce reliance on large models and strengthen privacy guarantees, we introduce Databox-Local, which trains small language models for Databox construction and enables fully local deployment without external LLM calls. Experiments on BIRD-CRITIC-PG, spanning five widely used agent scaffolds and three backbone LLMs, show that Databox reduces optimizer-estimated execution cost by 65%-92%, with an average reduction of 85.2% relative to real-database interaction, while maintaining comparable task performance and slightly increasing the average success rate by 1.4%. We further evaluate Databox on the repository-level benchmark BIRD-CRITIC-REPO, demonstrating its adaptability to broader software engineering settings.

1 INTRODUCTION

Relational databases remain the most established and widely adopted technology for the structured storage, organization, and retrieval of data, supporting core data-centric business operations across manufacturing, logistics, finance, and enterprise systems (Date, 2006; Hellerstein & Stonebraker, 2005). Structured query language (SQL) is the standard interface for interacting with relational databases (Chamberlin & Boyce, 1974). As database-driven businesses continue to evolve in scope and functional sophistication, the increasing semantic, syntactic, and operational complexity of SQL poses persistent challenges for users in query formulation, comprehension, and interpretation of database behavior (Chen et al., 2024; Kandel et al., 2012). This need motivates the study of SQL issue resolution, which focuses on refining problematic SQL statements based on user-reported issues and database context, thereby automating real-world SQL debugging to streamline LLM agent workflows and reduce dependence on expert intervention Li et al. (2025); Gathani et al. (2020).

Although large language model (LLM) agents have demonstrated strong capabilities in solving complex problems (Ouyang et al., 2022; Zhuo et al., 2024; Anthropic, 2025a), they often depend on dynamic interaction with executable environments to investigate and refine solutions (Li & Xie, 2024; Pourreza & Rafiei, 2023; Yang et al., 2024; Wang et al., 2024). Agents execute the problematic SQL to examine the reported behavior, run diagnostic queries to investigate possible causes, and execute proposed candidates to assess their effects. However, performing this process on real, large-scale databases introduces substantial execution costs, amplified by repeated queries across debugging

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

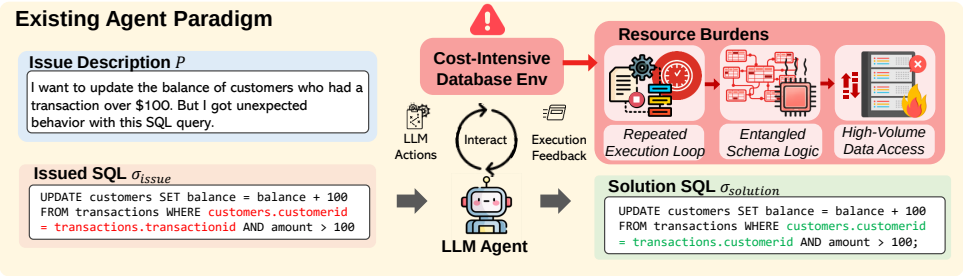


Figure 1: Agentic SQL debugging paradigm with real-database interaction. Repeated execution feedback loops, entangled schema logic, and high-volume data access impose heavy resource burdens.

iterations. Additionally, large schema descriptions and accumulated execution results increase the agent's context burden and exacerbate long-context reasoning challenges. Beyond these resource demands, direct database interaction can expose sensitive data and cause unintended changes to database state, bringing security and deployment risks (Awan et al., 2023; Qu et al., 2025). These limitations constrain the practical deployment of execution-guided agent methods in real-world SQL-based tasks.

In this paper, we introduce **Databox**, a lightweight, task-driven sandbox database environment for execution-guided SQL debugging. Our design is motivated by the observation that investigating a particular issue may require only a subset of schema components and specific data conditions relevant to the reported failure and the user's intent. Accordingly, Databox replaces real-database interaction with a compact, isolated environment constructed around these task-specific requirements. The challenge is to determine what this environment must retain, since omitting relevant structure or changing data can conceal the original defect or introduce misleading execution behavior. To address this challenge, Databox combines **task-specific schema extraction** (Section 3.1), **guideline-driven generation** of diagnostic values (Section 3.2), and **consistency validation** (Section 3.3) to reproduce the reported issue and support agent investigation and repair without accessing original records or modifying production state. We further introduce **Databox-Local**, which uses small language models to keep deployment-time construction inputs local and avoid external constructor API calls.

We evaluate Databox on BIRD-CRITIC-PG (Li et al., 2025) across five SQL agent scaffolds and three backbone models. Databox reduces optimizer-estimated execution cost by 65%–92%, averaging approximately 85%, while maintaining competitive debugging performance with an average success-rate gain of 1.4 percentage points. Measured SQL execution times broadly corroborate these savings, and further analyses show shorter execution feedback and fewer agent interactions in most evaluated settings. Beyond standalone SQL debugging, we extend the evaluation to repository-level software-engineering workflows through BIRD-CRITIC-REPO, where three general-purpose vibe-coding tools recover issue context from project files, modify SQL artifacts, and interact with databases through command-line interfaces and file systems. An additional Spider2-Lite SQLite study examines transfer to text-to-SQL and another SQL dialect. These results show that Databox consistently reduces execution costs across diverse backbone models, agentic frameworks, and SQL-level and repository-level task settings, while serving as a plug-in substitute for real databases.

2 Task Formulation

Environment-Grounded SQL Issue Resolution. Given a problematic SQL σ_{issue} and a natural-language issue description P, the goal of environment-grounded SQL issue resolution is to generate a corrected query σ_{pred} that fixes the reported issue while preserving user intent, formalized as

$$\sigma_{\text{pred}} = f_\theta(P, S, \sigma_{\text{issue}}; E), \tag{1}$$

where f_θ denotes the mapping function applied by the LLM agent, and $E = (S, V)$ denotes an executable database environment with schema S and values V. In conventional agentic settings, E corresponds to the real database $E_{\text{real}} = (S_{\text{real}}, V_{\text{real}})$, allowing agents to leverage authentic execution feedback for iterative refinement.

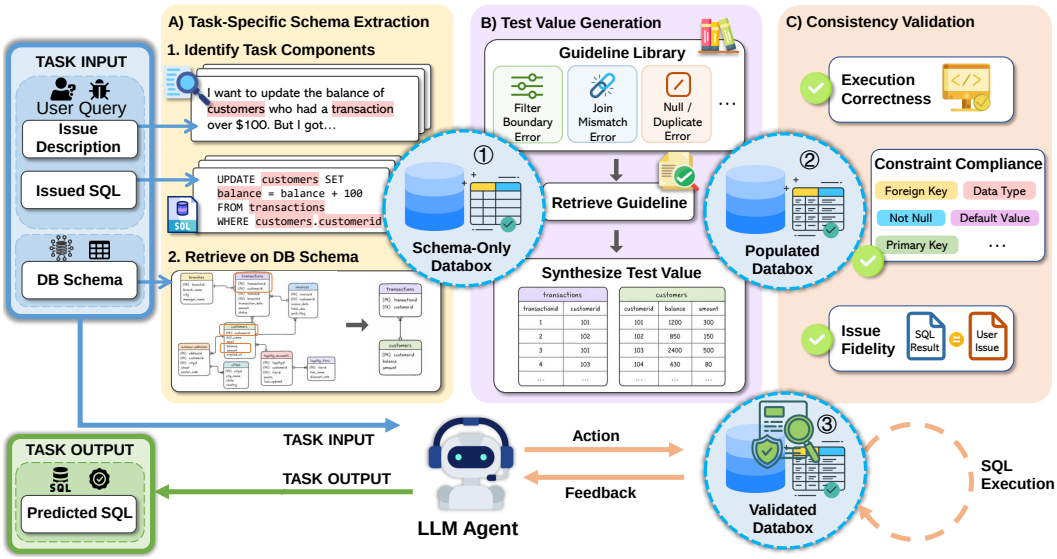


Figure 2: Overview of Databox. The upper panel illustrates the three-stage construction procedure: task-specific schema extraction, test value generation, and consistency validation. The lower panel presents the agent workflow for SQL issue solving with Databox.

However, access to the real database E_{real} is often impractical due to safety limitations or resource burden constraints. We therefore introduce Databox $E' = (S', V')$ as an alternative execution environment, where $S' \subseteq S_{\text{real}}$ contains task-relevant schema and V' consists of generated test values. Given the problematic SQL, the issue description, and the original schema, our goal is to construct a compact environment that reproduces the reported behavior and supports execution-based investigation and repair, especially for privacy-sensitive and resource-constrained scenarios.

3 Databox Construction

Databox constructs a compact execution environment for each SQL issue by retaining only the schema structure and data conditions needed for debugging. The key challenge is to simplify the environment without changing the execution feedback needed to expose the reported problem, since missing relations, constraints, or diagnostic values can conceal the original defect or make an incorrect repair appear valid. As illustrated in Figure 2, Databox addresses this challenge through three stages. Task-specific schema extraction identifies relevant schema components using the problematic SQL, the issue description, and the original schema (Section 3.1). Guideline-driven test value generation synthesizes data conditions intended to expose the reported issue within the selected schema (Section 3.2). Consistency validation checks construction executability, constraint compliance, and reproduction of the reported behavior, triggering further construction attempts when checks fail (Section 3.3).

3.1 Task-Specific Schema Extraction.

The schema extraction stage constructs a compact schema $S' \subseteq S_{\text{real}}$ that retains the tables, columns, types, and constraints needed to interpret and execute task-relevant SQL. However, insufficient pruning leaves irrelevant schema information that increases avoidable complexity and execution overhead, whereas excessive pruning can remove essential components and prevent candidate repairs from being properly evaluated through execution. The objective is therefore to reduce schema complexity while retaining the structural context required for debugging.

The difficulty is that neither the problematic SQL nor the issue description alone fully specifies the required schema, making schema component selection necessary to consider both jointly. The problematic SQL contains explicit object references but may omit components necessary for the

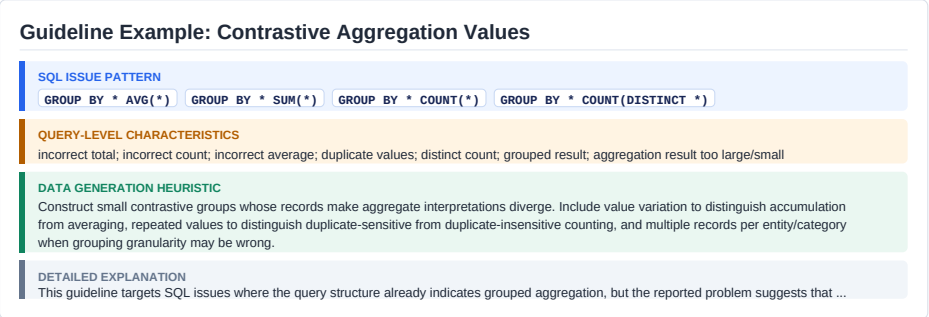


Figure 3: Example of a diagnostic data-generation guideline for grouped aggregation issues. The guideline combines SQL issue patterns, query-level characteristics, and a construction heuristic, guiding the construction of compact groups with contrasting values and duplicates to expose incorrect aggregate functions or distinct-count semantics.

intended operation, while the description conveys user intent using natural-language expressions that can differ from the names used in the database schema. We adopt a dual-source schema extraction approach in which candidates are separately identified from the SQL through rule-based extraction and from entities mentioned in the description. An LLM later integrates and supplements these candidates by considering the original database metadata, then generates executable DDL statements to construct the schema-only environment $E' = (S', \emptyset)$.

Additionally, enforcing integrity constraints before data population can impose restrictive insertion orders. We therefore adopt staged constraint enforcement, retaining constraint specifications during schema extraction while postponing the installation of selected constraints until after data insertion. This allows greater flexibility during the data-generation stage while satisfying the retained constraints.

3.2 Test Value Generation.

After constructing the schema-only environment, we populate it with diagnostic values V'. Direct sampling risks exposing real records and requires additional processing to preserve dependencies and integrity constraints within the reduced schema. We instead generate values under the selected schema, avoiding real-record retrieval and adaptation. The core task is translating the problematic SQL and issue description into values and row relationships that trigger the reported behavior. Our guideline-driven approach provides reusable construction strategies to guide this process.

As illustrated in Figure 3, each guideline specifies a SQL pattern, applicable issue characteristics, and a data-construction heuristic. For a new issue instance, we retrieve candidate guidelines according to its SQL structure and issue pattern and incorporate them into the generation prompt. Guided by these heuristics and the previously constructed schema S', an LLM generates diagnostic values V' and assembles them into executable data manipulation language (DML) statements.

Guidelines are prepared offline and stored in a library through exploration and ranking stages. During exploration, an LLM examines the problematic and reference SQL to reason backward from their differences and propose diagnostic values with corresponding construction guidelines. Candidates are retained when their values reproduce the reported issue and distinguish the two queries' execution outcomes. During ranking, an LLM applies candidate guidelines to additional issue instances without seeing their solutions. Guidelines are ranked by the number of successful instances, and the highest-ranked candidates form the final library. Further details are provided in Appendix A.

3.3 Consistency Validation.

Schema and data that appear correct individually can be incompatible when combined or fail to reproduce the reported issue. Therefore, we validate the complete environment through three checks: (1) **Construction executability** verifies that the initialization statements execute successfully. (2) **Constraint compliance** checks that the populated data satisfies the retained integrity constraints. (3)

Issue fidelity assesses whether executing the problematic SQL exhibits the problematic behavior described by the user.

If any check fails, Databox iteratively revises the construction by revisiting the relevant stages. However, the observed problem may originate from either component or their interaction, so both components should be considered together. We therefore adopt a coupled reconstruction strategy for LLM-based Databox revision. A heuristic checker determines whether to regenerate a specific stage or both, and packages previous construction artifacts and validation diagnostics as regeneration context. Thus, each component is revised with awareness of the other component and the validation feedback. Only Databoxes passing all three checks are retained for subsequent SQL debugging.

3.4 DATABOX-LOCAL: LOCAL CONSTRUCTION WITH SLMS

Although Databox avoids retrieving original records, external LLM construction still exposes schema and issue information to the provider and incurs API costs. We therefore introduce Databox-Local, which trains two small language models (SLMs) for schema extraction and diagnostic test value generation to keep the construction process local and avoid external LLM API calls.

Discriminative data collection. Training local Databox constructors requires examples of environments that expose SQL issues. We first conduct QEP-driven modifications to inject issues into correct SQL queries from text-to-SQL data, producing paired problematic and reference SQL. A teacher LLM generates construction trajectories, from which we retain environments that reproduce the reported issue and yield different execution outcomes for the two SQL statements. These environments distinguish the defect from its reference repair, providing supervision for diagnostic construction. This selection specializes the SLMs toward diagnostic environments that expose SQL defects. Dataset preparation, trajectory selection, and local validation are detailed in Appendix C.

Training objectives. For each issue instance, the common input is $x = (P, \sigma_{\text{issue}}, S_{\text{real}})$, consisting of the issue description, problematic SQL, and original schema. We decompose construction into two supervised generation tasks. (1) **Schema extraction** produces $y_s = (t_s, D_s)$, where t_s is the schema-selection rationale and D_s contains executable DDL statements that construct the schema-only environment $E' = (S', \emptyset)$. (2) **Test value generation**, additionally conditioned on the constructed schema S', produces $y_v = (r_v, D_v)$, where r_v is the diagnostic guideline and D_v contains executable DML statements that populate the environment with test values V'. Both SLMs are based on QWEN2.5-CODER-7B-INSTRUCT and trained using LoRA fine-tuning (Hu et al., 2021).

3.5 INTERFACE DESIGN

To support diverse application scenarios and facilitate adoption in existing LLM agent frameworks, Databox provides two types of standardized interfaces: (1) **Tool-based interface**: This supports a text-based action interface that accepts raw SQL queries as agent actions and returns execution results as observations, as well as a Python API that exposes Databox functionality through function calls. (2) **CLI-based interfaces**: To align with terminal-centric agent workflows and vibe-coding tools, Databox supports database-style CLI interaction similar to conventional systems such as `PostgreSQL`. The file-based interface further allows agents to read and execute SQL statements and write execution results through the file system, making it suitable for multi-file debugging and repository-level reasoning in software engineering environments.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks and Metrics. We evaluate SQL issue resolution on BIRD-CRITIC-PG (Li et al., 2025), a PostgreSQL benchmark comprising 530 instances across 15 databases and eight issue categories. We further evaluate repository-level debugging on BIRD-CRITIC-REPO, which embeds the issues in multi-file codebases.

We report results along two key dimensions: success rate (SR) and estimated execution cost. Estimated execution cost is obtained from the optimizer-estimated total cost in the QEP, which reflects the

Table 1: Main experiment results on BIRD-CRITIC-PG. SR% denotes success rate, and Est.Cost denotes optimizer-estimated execution cost from QEPs.

Method	Variant	Gemini-2.0-Flash		Claude-3.7-Sonnet		Qwen3-Coder-480B-A35B-Instruct	
		SR%	Est.Cost	SR%	Est.Cost	SR%	Est.Cost
End-to-end methods							
Simple Prompt	-	29.3	-	32.1	-	34.9	-
Chain-of-Thought	-	30.2	-	28.1	-	34.0	-
Agentic SQL methods							
SQL-Act	+Real DB	36.6	1947	37.0	248	40.7	664
	+Databox	40.6 ↑(+4.0)	370 ↓(-81%)	38.3 ↑(+1.3)	57 ↓(-77%)	41.7 ↑(+1.0)	60 ↓(-91%)
IntercodeSQL	+Real DB	33.6	3152	35.3	447	37.9	1914
	+Databox	35.5 ↑(+1.9)	314 ↓(-90%)	36.0 ↑(+0.8)	98 ↓(-78%)	41.4 ↑(+3.5)	266 ↓(-86%)
Spider2Agent	+Real DB	36.2	3233	37.7	1498	38.7	945
	+Databox	37.0 ↑(+0.8)	248 ↓(-92%)	38.9 ↑(+1.1)	186 ↓(-88%)	39.1 ↑(+0.4)	95 ↓(-90%)
AutoGPT-SQL	+Real DB	29.5	2655	29.3	805	35.5	722
	+Databox	31.9 ↑(+2.4)	339 ↓(-87%)	29.8 ↑(+0.5)	117 ↓(-86%)	34.8 ↓(-0.6)	253 ↓(-65%)
CHASESQL	+Real DB	34.2	2392	32.1	967	33.4	725
	+Databox	36.4 ↑(+2.3)	268 ↓(-89%)	32.6 ↑(+0.6)	123 ↓(-87%)	34.0 ↑(+0.6)	68 ↓(-91%)

planner's cumulative estimate of I/O and CPU workload. Unlike wall-clock execution time, this metric is less sensitive to transient system load and runtime fluctuations, thereby enabling more stable comparisons. Further analyses examine real SQL execution time, agent action effort, execution-feedback length, and model token usage.

Methods. We consider both end-to-end prompting methods and agentic approaches. **End-to-end prompting** directly generates the corrected SQL without any interaction. We evaluate simple zero-shot prompting and chain-of-thought prompting. **SQL agentic methods** support execution-based SQL debugging and iterative self-reflection. We evaluate five commonly used SQL agent scaffolds, including SQL-ACT (Li et al., 2025), Intercode-SQL (Yang et al., 2023), Spider2Agent (Lei et al., 2024), AutoGPT-SQL, and CHASESQL (Pourreza et al., 2024), spanning SQL issue-solving agents, text-to-SQL agents, and general code-oriented agents adapted to SQL debugging.

Models and Configurations. We conduct experiments with GEMINI-2.0-FLASH, CLAUDE-3.7-SONNET (Anthropic, 2025), and QWEN3-CODER-480B-A35B-INSTRUCT (Yang et al., 2025). For agents originally designed for text-to-SQL or general code generation, we make only minimal task-specific adaptations by rewriting prompts for SQL issue resolution and exposing the required SQL execution tools. In the Databox condition, an environment is constructed for each issue before debugging and serves as the agent's sole database execution target.

4.2 RESULTS ON SQL-LEVEL ISSUE SOLUTION

Table 1 presents the results on BIRD-CRITIC-PG, comparing execution-free end-to-end prompting baselines with execution-guided SQL agents under both real-database interaction and Databox substitution. We summarize the main observations below.

Execution-guided agents generally achieve higher success rates across the evaluated backbones. One-shot end-to-end methods are generally unreliable and achieve lower performance, as they rely solely on static internal reasoning and lack execution-based guidance. In contrast, agentic workflows address this bottleneck with superior performance by incorporating execution-based feedback into the debugging process. Executable environments allow agents to investigate reported behavior, test possible causes, and assess candidate revisions through observed results. These findings motivate preserving execution-guided interaction when addressing the costs and risks of real-database access, which is the objective of Databox.

Table 3: Repository-level experiment results on BIRD-CRITIC-REPO. SR% denotes success rate, and Est.Cost denotes optimizer estimated execution cost.

Method	Variant	Gemini-2.0-Flash		Claude-3.7-Sonnet		Qwen3-Coder-480B-A35B-Instruct	
		SR%	Est.Cost	SR%	Est.Cost	SR%	Est.Cost
OpenCode	+Real DB	30.75	1389	37.17	257	44.53	831
	+Databox	33.96 ↑(+3.21)	79 ↓(-94%)	41.70 ↑(+4.53)	33 ↓(-87%)	41.13 ↓(-3.40)	43 ↓(-95%)
QwenCode	+Real DB	30.19	1083	31.32	947	45.09	1168
	+Databox	33.40 ↑(+3.21)	64 ↓(-94%)	34.15 ↑(+2.83)	127 ↓(-87%)	42.45 ↓(-2.64)	62 ↓(-95%)
KimiCode	+Real DB	29.25	328	42.26	73	47.92	203
	+Databox	32.26 ↑(+3.01)	60 ↓(-82%)	39.81 ↓(-2.45)	19 ↓(-74%)	44.34 ↓(-3.58)	33 ↓(-84%)

Databox significantly reduces execution cost while preserving solution quality. Databox achieves significant cost reduction across all evaluated agentic frameworks and backbone models compared to real-database interaction. Specifically, Databox reduces estimated execution cost by 65%-92% across agents, with an average reduction of 85%. Measured SQL execution times broadly corroborate the optimizer-cost reductions; full runtime distributions and measurement details are reported in Appendix H. Without access to real data, Databox consistently preserves success rates relative to real databases and even achieves slight improvements. These modest gains partly reflect more focused debugging behavior within the task-specific environment. We examine this interpretation through interaction counts and action distributions in Section 4.6.

4.3 Results of Databox-Local

We evaluate whether locally deployed small language models can construct effective Databox environments. We use Gemini-2.0-Flash as the debugging-agent backbone across four scaffolds. Table 2 compares RealDB interaction with environments constructed by Databox-LLM and Databox-Local. Databox-Local achieves success rates close to the LLM constructor for SQL-ACT, IntercodeSQL, and AutoGPT-SQL, with differences below 0.5 percentage points, while only Spider2Agent shows a larger decrease, from 37.00% to 31.89%. Despite this variation, Databox-Local reduces optimizer-estimated execution cost by 75.24%–92.39% relative to RealDB, retaining the substantial execution savings of Databox, which is on par with LLM-constructed Databox. These results demonstrate the feasibility of constructing Databoxes using local SLMs in privacy-sensitive or deployment-constrained settings, without relying on external LLM interactions.

Table 2: Databox-Local experiment results on BIRD-CRITIC-PG with Gemini-2.0-Flash.

Method	**Variant**	**SR%**	**Est.Cost**	**↓%**
SQL-Act	+Real DB	36.60	1947	–
	+Databox-LLM	40.60	370	81.00%
	+Databox-Local	40.38	482	75.24%
IntercodeSQL	+Real DB	33.60	3152	–
	+Databox-LLM	35.50	314	90.04%
	+Databox-Local	35.09	542	82.80%
Spider2Agent	+Real DB	36.20	3233	–
	+Databox-LLM	37.00	248	92.33%
	+Databox-Local	31.89	295	90.88%
AutoGPT-SQL	+Real DB	29.50	2655	–
	+Databox-LLM	31.90	339	87.23%
	+Databox-Local	31.70	202	92.39%

4.4 Generalization Across Repository-Level Workflows and Text-to-SQL Tasks

Extension to repository-level Debugging. We evaluate Databox on BIRD-CRITIC-REPO, where SQL issues are embedded in multi-file code repositories and agents must recover issue context, edit SQL artifacts, and execute database commands through the CLI. We evaluate three widely used vibe-coding tools for this task: OpenCode, KimiCode, and QwenCode. Table 3 shows that Databox reduces estimated execution cost by 74%–95% across three vibe-coding tools and backbone models, while keeping success-rate changes within 3.6 percentage points. These results indicate that Databox can serve as a plug-in execution environment for repository-level coding workflows beyond SQL-specialized agents.

Table 4: Effect of data population strategy on Databox construction and SQL repair.

Data generation strategy	Constructed / 505	Official Pass / 505	Primary SR	Conditional SR
Databox, no preview	488	**190**	**37.62%**	**38.93%**
Preview-row copy	505	180	35.64%	35.64%
Databox + preview	458	172	34.06%	37.55%
Bernoulli sample + FK closure	505	179	35.45%	35.45%
DOMINO (SchemaAnalyst)	420	149	29.50%	35.48%
SDV-HMA	194	74	14.65%	38.14%
CTGAN + relational adapter	360	132	26.14%	36.67%

Extension to text-to-SQL and SQLite. We further evaluate Databox on 106 Spider2-Lite SQLite instances, extending beyond PostgreSQL debugging to text-to-SQL. Databox achieves 10.55–$21.25\times$ speedups in median final-query execution time while retaining broadly comparable task accuracy, with success-rate differences ranging from -5.66 to $+3.77$ percentage points relative to RealDB. Detailed results and configuration-specific comparisons are provided in Appendix M.

4.5 Data Generation Strategy Ablation

This ablation examines whether Databox's issue-guided data generation produces more effective debugging environments than simple sampling or using general-purpose data generators. We evaluate population strategies on 505 BIRD-CRITIC-PG instances with SQL-ACT and Gemini-2.5-Flash, keeping the schema and debugging protocol fixed. Table 4 reports successful constructions, overall repair success, and conditional success among constructed environments.

Databox achieves the highest overall success rate of 37.62%, compared with 35.64% for preview-row copying and 35.45% for Bernoulli sampling, despite both alternatives successfully populating all instances. General-purpose generators achieve lower overall success, partly because of construction failures. These results support issue-guided generation as an effective population strategy for SQL debugging, where both successful construction and diagnostic data selection matter. Detailed configurations and analysis are provided in Appendix I.

4.6 Databox Improves Efficiency via Focused Debugging

Databox reduces agent interaction effort. We further investigate the sources of Databox's efficiency gains beyond cheaper SQL execution in a smaller database environment. Average action counts per task decrease in seven of nine SQL-agent settings, with reductions reaching 28.4%. In repository-level experiments, where vibe-coding tools determine when to stop without a fixed action-turn cap, average command counts decrease by 26.7%–28.0% and recovered PostgreSQL statement counts decrease by 22.7%–25.6% across the three model backbones. These results suggest that Databox's execution efficiency benefits arise from both cheaper individual queries and fewer repeated interactions during task completion. Detailed results and configuration-specific behavior are discussed in Appendix J.

Databox leads to a more decisive action distribution. Appendix J shows that agents using Databox devote a smaller proportion of actions to inspection and partial SQL solutions, with greater emphasis on directly testing candidate repairs. Together with the action-count results, this pattern suggests that the task-specific environment reduces exploratory overhead and makes debugging more focused. Execution-guided agents use observed results to diagnose problems and revise SQL iteratively; concentrating these interactions on candidate repairs may help them use their debugging budget more effectively. This may help explain the modest success-rate gains observed in the main experiments.

4.7 Shorter Feedback and Isolated Database Modifications

Less context from execution observations. Databox reduces mean feedback length by 5.9%–69.3% across all nine agent–model settings, decreasing the context introduced by individual execution

observations. Reductions persist after excluding the 5% extreme values. Detailed measurements and construction and interaction token overheads are reported in Appendices K and G, respectively.

Correct answers can conceal unauthorized database changes. Trajectory-level analysis identifies unauthorized database modifications in 18.0% of the sampled RealDB trajectories; database modifications also occur in 19.7% of Databox trajectories. Of the 54 RealDB trajectories with unauthorized modifications, 32 nevertheless produce correct final solutions and are considered passed. Final-answer correctness therefore leaves an important operational risk unmeasured: agents can return a valid repair while making unauthorized changes to database state. Databox confines exploratory modifications to its isolated environment, protecting production state during debugging. The trajectory analysis is provided in Appendix L.

5 Related work

5.1 From Text-to-SQL To SQL Issue Resolution

Text-to-SQL aims to translate natural language queries into executable SQL statements, enabling non-expert users to interact with relational databases without requiring technical expertise (Zelle & Mooney, 1996; Yu et al., 2018; Androutsopoulos et al., 1995). Recent advances in large language models (LLMs) have significantly improved performance on popular benchmarks, such as SPIDER (Yu et al., 2018) and BIRD (Li et al., 2023), demonstrating strong capabilities in semantic parsing and structured query generation. As real-world data systems continue to grow in scale and complexity, there is an increasing demand to extend Text-to-SQL beyond traditional settings toward more realistic and comprehensive data-centric tasks (Chen et al., 2024; Li et al., 2023; 2024). Spider2 evaluates model performance on enterprise-level databases with complex schemas and large data volume (Lei et al., 2024). SWE-SQL introduces real-world SQL issue resolution scenarios, expanding the task scope from SELECT-only queries to general operations (Li et al., 2025). BIRD-INTERACT further emphasizes interactive and dynamic Text-to-SQL settings, requiring interactive ambiguity resolution, error recovery, and CRUD operations (Huo et al., 2026). These developments highlight a shift from canonical Text-to-SQL tasks toward more realistic settings that require LLMs to handle complex query logic, heterogeneous database scales, and significant resource burdens.

5.2 LLM for SQL-based Tasks

LLMs have achieved strong performance in coding, especially SQL-related tasks (Chen et al., 2021; Ouyang et al., 2022; Kojima et al., 2022). Recent work has shown agentic frameworks to be effective for SQL-based tasks, leveraging iterative interaction with database environments (Qin et al., 2022; Zhang et al., 2025). Representative methods such as CHASE-SQL employ generate-execute-validate pipelines and select the best candidate (Pourreza et al., 2024), while other advanced agents like SQL-ACT (Li et al., 2025) and Spider2Agent (Lei et al., 2024) incorporate structured tools and database actions to support complex workflows, including schema inspection, query debugging, and database modification. These approaches adopt execution-guided self-reflection and reasoning, where LLMs generate, execute, and refine SQL queries based on feedback signals such as execution results or error messages, leading to substantial performance improvements.

6 Conclusion

We introduce Databox, a lightweight, task-specific sandbox environment that supports effective LLM agent interaction for SQL issue resolution. Databox is constructed through task-specific schema extraction, test value generation, and consistency validation, serving as a plug-in module that replaces costly and potentially sensitive real-database interactions. We further develop an SLM-based Databox constructor, Databox-Local, designed for stricter privacy constraints and to avoid LLM API costs. Experiments on BIRD-CRITIC-PG show an average execution-cost reduction of 85% across five agentic scaffolds and three backbone models. Further experiments on BIRD-CRITIC-REPO and Spider2-Lite SQLite demonstrate Databox's generalization potential for broader software engineering and text-to-SQL settings.

AI USE STATEMENT

In this work, we used generative AI tools for writing assistance, grammar checking, coding assistance, and experiments with different models, as detailed in the experimental setup. We did not use generative AI tools for idea generation, literature search, or figure/table generation. We reviewed all AI-assisted writing and code, verified experimental outputs through our evaluation pipeline, and take full responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work aims to make LLM-based SQL debugging agents safer and more practical when direct production-database access is costly, unavailable, or inappropriate. Databox reduces repeated real-database interaction by using compact sandbox environments with synthetic diagnostic values, and Databox-Local further supports deployment without external LLM API calls. However, Databox is a debugging substitute rather than a replacement for production validation: incorrect sandbox construction may miss important constraints or edge cases, and corrected SQL should be reviewed before deployment, especially in high-stakes domains. Our experiments use publicly available benchmark data rather than private production databases, and generated Databox values are synthetic. If applied to proprietary schemas, users should still enforce appropriate access control, logging, query auditing, and human approval for schema or data modifications, since schema metadata and agent actions may carry security or governance risks.

REPRODUCIBILITY STATEMENT

We provide the task formulation, Databox construction procedure, training-data construction, model settings, evaluation protocol, and implementation details in the main paper and appendix. The experimental setup describes the evaluated benchmarks, agent scaffolds, backbone models, decoding settings, interaction budgets, and metrics, while Appendix D details the execution-cost computation, model implementation, and repository-level benchmark construction. Appendix A provides the guideline-driven test-value generation procedure, Appendix B describes QEP-driven issue injection, and Appendix N lists the prompt templates used for Databox construction and repository-level evaluation.

REFERENCES

Abdullah Alsharif, Gregory M Kapfhammer, and Phil McMinn. Domino: Fast and effective test data generation for relational database schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 12–22. IEEE, 2018.

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81, 1995.

Anthropic. Claude sonnet, 2025. URL `https://www.anthropic.com/claude/sonnet`.

Anthropic. Claude code, 2025a. URL `https://claude.com/product/claude-code`.

Kamran Ahmad Awan, Ikram Ud Din, Ahmad Almogren, and Joel JPC Rodrigues. Privacy-preserving big data security for iot with federated learning and cryptography. *IEEE access*, 11:120918–120934, 2023.

Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pp. 249–264, 1974.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, 1998.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Peter Baile Chen, Fabian Wenz, Yi Zhang, Devin Yang, Justin Choi, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. Beaver: an enterprise benchmark for text-to-sql. *arXiv preprint arXiv:2409.02038*, 2024.

Christopher John Date. *An introduction to database systems*. Pearson Education India, 2006.

Sneha Gathani, Peter Lim, and Leilani Battle. Debugging database queries: A survey of tools, techniques, and users. In *proceedings of the 2020 CHI conference on human factors in computing systems*, pp. 1–16, 2020.

Joseph M Hellerstein and Michael Stonebraker. *Readings in database systems*. MIT press, 2005.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, et al. Bird-interact: Re-imagining text-to-sql evaluation via lens of dynamic interactions. In *The Fourteenth International Conference on Learning Representations*, 2026.

Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE transactions on visualization and computer graphics*, 18 (12):2917–2926, 2012.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. The dawn of natural language to sql: Are we fully ready? *arXiv preprint arXiv:2406.01265*, 2024.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, et al. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications. *arXiv preprint arXiv:2506.18951*, 2025.

Zhenwen Li and Tao Xie. Using llm to select the right sql query from candidates. *arXiv preprint arXiv:2401.02115*, 2024.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE international conference on data science and advanced analytics (DSAA)*, pp. 399–410. IEEE, 2016.

Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in neural information processing systems*, 36:36339–36348, 2023.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*, 2024.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, et al. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*, 2022.

Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. Share: An slm-based hierarchical action correction assistant for text-to-sql. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11268–11292, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in neural information processing systems*, 32, 2019.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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*, pp. 3911–3921, 2018.

Algorithm 1 Guideline Library Construction

Step 1: Guideline Exploration

Input: Seed instances $\mathcal{D}_{\text{seed}}$, exploration rounds M

Initialize $\mathcal{R}_{\text{cand}} \leftarrow \emptyset$

for all $(P_i, \sigma_{\text{issue}}^{(i)}, S_i, \sigma_i^*) \in \mathcal{D}_{\text{seed}}$ **do**
  for $j = 1, \ldots, M$ **do**
    Generate (r_{ij}, V_{ij})
    $E_{ij} = (S_i, V_{ij})$
    if $\text{Exec}(\sigma_{\text{issue}}^{(i)}; E_{ij}) \neq \text{Exec}(\sigma_i^*; E_{ij})$ and V_{ij} consistent with P_i
      $\mathcal{R}_{\text{cand}} \leftarrow \mathcal{R}_{\text{cand}} \cup \{r_{ij}\}$
    end if
  end for
end for

return $\mathcal{R}_{\text{cand}}$

Step 2: Guideline Evaluation and Ranking

Input: Candidate guidelines $\mathcal{R}_{\text{cand}}$, evaluation instances $\mathcal{D}_{\text{eval}}$

for all $r \in \mathcal{R}_{\text{cand}}$ **do**
  Randomly match evaluation instances $\mathcal{D}_{\text{eval}}(r)$
  $\text{Score}(r) \leftarrow 0$
  for all $(P_i, \sigma_{\text{issue}}^{(i)}, S_i, \sigma_i^*) \in \mathcal{D}_{\text{eval}}(r)$ **do**
    Generate test values $V_{i,r}$ by applying h_r
    $E_{i,r} = (S_i, V_{i,r})$
    if $\text{Exec}(\sigma_{\text{issue}}^{(i)}; E_{i,r}) \neq \text{Exec}(\sigma_i^*; E_{i,r})$ and $V_{i,r}$ consistent with P_i
      $\text{Score}(r) \leftarrow \text{Score}(r) + 1$
    end if
  end for
end for
$\mathcal{R} \leftarrow \text{TopK}_{r \in \mathcal{R}_{\text{cand}}} \text{Score}(r)$
return $\mathcal{R}$

John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pp. 1050–1055, 1996.

Xuanliang Zhang, Dingzirui Wang, Longxu Dou, Qingfu Zhu, and Wanxiang Che. A survey of table reasoning with large language models. *Frontiers of Computer Science*, 19(9):199348, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A GUIDELINE-DRIVEN TEST VALUE GENERATION

A guideline consists of: (1) an applicable SQL issue pattern (e.g., missing `JOIN` condition, incorrect aggregation scope, misordered `WHERE` clause), (2) the associated query-level characteristics that trigger this pattern, and (3) a data generation heuristic specifying how to construct test values and expose the error. By capturing common SQL error patterns and their corresponding diagnostic value construction heuristics, guidelines can be concluded through seed datasets and reused across instances with similar query characteristics.

Guideline Exploration. We sample 300 seed instances from the SWE-GYM(Li et al., 2025) together with their task-specific schema-only database. Each instance containing an issue SQL σ_{issue}, a user-reported issue description P, the schema-only database S, and a ground-truth correct SQL σ^*. The LLM, GEMINI-2.0-FLASH, are prompted to generate test values V together with an explicit natural-language guideline r_i describing the underlying SQL issue pattern p_r, the reported issue characteristics c_r, and the rationale for the constructed data h_r, where $r = (p_r, c_r, h_r)$. A candidate guideline is retained only if the generated test values produce different execution outcomes for the issue SQL and the correct SQL and faithfully recreate the reported issue. To encourage diversity, the LLM is explicitly forbidden to reuse previously discovered guidelines across multiple generation rounds, and each seed instance is used in three distinct guideline exploration rounds. This process collects a pool of 130 candidate guidelines, each validated on at least one seed instance.

Guideline Ranking. To select guidelines with higher generalizability, we conduct an evaluation and ranking process using the remaining instances. in SWE-GYM. For each guideline r, we randomly match issue instances to its applicability conditions based on SQL patterns and issue characteristics, and then apply the guideline to generate test values. Each guideline is scored and ranked by its empirical coverage, defined as the number of matched evaluation instances for which it produces valid

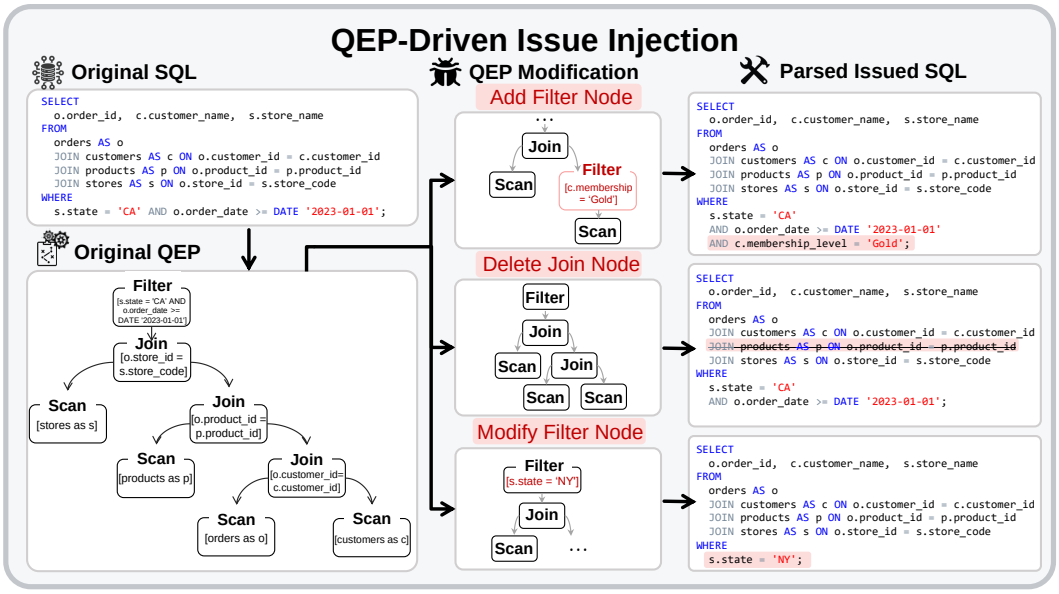


Figure 4: QEP-driven issue injection. We apply three types of QEP modification: Node Addition, Node Deletion, and Node Attribute Modification. Node Addition introduces extra operations, Node Deletion removes necessary logic, and Node Attribute Modification alters node functionality, e.g., changing a filter condition.

diagnostic test values. Through sufficient instance sampling and evaluation, the top-50 guidelines are retained as a high-quality guideline library. Algorithm 1 summarizes the guideline library construction procedure.

Guideline Retrieval in Databox Construction. In the test value generation stage, given a new SQL issue instance, Databox constructor retrieves a small subset of guidelines from the ranked guideline library. The retrieval is guided by matching the structural features of the issue SQL pattern (e.g., query type, clause composition, join structure) and the semantics of the user-reported issue description against the applicability conditions of each guideline. Typically, 1–3 top-ranked guidelines are selected and incorporated into the prompt to guide the test value generation, with concrete heuristics for constructing diagnostic data tailored to the target issue.

B Analysis of QEP-driven Issue Injection

B.1 QEP Issue Generation Types

QEP-driven issue injection modifies the query execution plan before converting the modified plan back into an executable SQL query. We consider three simple modification types, illustrated in Figure 4. **Node Addition** inserts an extra plan operation, such as an unnecessary filter, join, aggregation, or ordering step, which changes the query behavior by adding logic that should not exist. **Node Deletion** removes a necessary plan operation, such as a required filter condition, join condition, grouping operation, or projection step, causing the generated SQL to miss part of the intended semantics. **Node Attribute Modification** keeps the plan structure but changes the functionality of an existing node, for example by altering a filter predicate, aggregation function, join key, ordering direction, or limit value. These three operations allow us to inject issues at the execution-logic level rather than only editing surface SQL tokens.

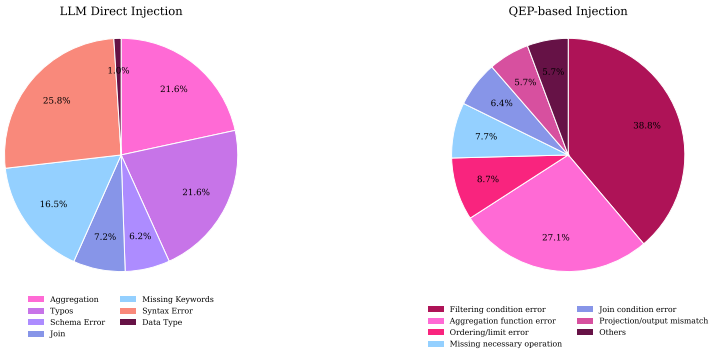


Figure 5: Issue-type distribution comparison between direct LLM-based issue injection and QEP-based issue injection.

Injection method	Issue type	Percentage
LLM direct injection	Direct injection	21.6%
	Aggregation	21.6%
	Typos	7.2%
	Schema error	6.2%
	Join	16.5%
	Missing keywords	25.8%
	Syntax error	1.0%
QEP-based injection	Filtering condition error	38.8%
	Aggregation function error	27.1%
	Ordering/limit error	8.7%
	Missing necessary operation	7.7%
	Join condition error	6.4%
	Projection/output mismatch	5.7%
	Others	5.7%

Table 5: Issue-type breakdown for direct LLM injection and QEP-based injection.

B.2 Issue Type Distribution

Figure 5 compares the issue-type distributions produced by direct LLM-based issue injection and QEP-based issue injection. The two construction strategies exhibit complementary behavior. Direct LLM injection produces a broader spread of surface-level SQL issues, including missing keywords, basic aggregation mistakes, typos, schema errors, join errors, and syntax errors. In contrast, QEP-based injection is more concentrated on execution-logic errors, especially filtering-condition errors and aggregation-function errors, which together account for the majority of QEP-injected cases. This pattern supports our motivation for introducing QEP-driven injection: directly modifying SQL text tends to create easier syntactic or local-edit issues, whereas modifying the execution plan can trigger deeper errors in the underlying query logic.

The distribution also reveals different roles for the two injection strategies. LLM direct injection is useful for quickly generating syntactic, schema-level, and local SQL-edit errors. Its largest categories are missing keywords (25.8%), direct injection (21.6%), and aggregation (21.6%), suggesting that free-form SQL corruption often affects visible query tokens or common SQL clauses. QEP-based injection, by contrast, produces fewer but more semantically targeted categories. Filtering condition errors (38.8%) and aggregation function errors (27.1%) dominate, indicating that plan-level modification frequently changes the semantics of row selection or value aggregation rather than merely corrupting surface syntax.

These results suggest that the two methods should be viewed as complementary rather than interchangeable. Direct LLM injection covers simpler and more surface-level mistakes that are common

in SQL editing, while QEP-based injection is essential for stress-testing deeper execution correctness, including filtering logic, aggregation behavior, ordering or limit semantics, join conditions, and output projection. Combining both strategies provides broader coverage over the SQL debugging space, from local syntactic errors to execution-level reasoning failures.

C DATABOX-LOCAL TRAINING AND CONSTRUCTION

C.1 QEP-DRIVEN TRAINING INSTANCE PREPARATION

Training local constructors requires SQL issue instances with reference solutions that can be used to assess generated environments. We use the BIRD-SQL development set (Li et al., 2023) as the source text-to-SQL dataset, taking each natural-language request and its correct SQL as a starting point. We inject a defect into the correct SQL to obtain a problematic query while retaining the unmodified query as its reference solution. This produces paired problematic and reference SQL for evaluating the diagnostic value of constructed environments. The associated natural-language request is rewritten as an issue report, yielding an instance containing the report, problematic SQL, original schema, and reference solution.

To introduce defects in query execution logic, we use query execution plan (QEP)-driven issue injection. A QEP describes the physical operators used by a database optimizer to execute a query (Chaudhuri, 1998). We extract the reference query's QEP, modify selected operators, and convert the modified plan back into SQL. We consider three modification types. **Node Addition** introduces an operation, such as a filter, join, or aggregation. **Node Deletion** removes an operation required by the original query. **Node Attribute Modification** changes an existing operator's behavior, such as a predicate, aggregation function, or join key. These modifications control the location and type of injected defects at the plan level. Examples and the resulting issue-type distribution are presented in Appendix B.

After issue injection, GEMINI-2.0-FLASH rewrites the original request into an issue-report-style description using predefined exemplars. Following the validation procedure of SWE-SQL (Li et al., 2025), we check consistency among the problematic SQL, the original reference solution, and the issue description. This process produces 5,000 SQL issue–resolution instances for preparing construction supervision.

C.2 DISCRIMINATIVE CONSTRUCTION TRAJECTORY SELECTION

We use GEMINI-2.0-FLASH to generate Databox construction trajectories for the prepared issue instances. The desired environment reproduces behavior consistent with the issue report and makes the discrepancy between the problematic and reference SQL observable. An environment in which both queries produce the same outcome provides no execution-based distinction between that defect and its reference repair. Selecting environments with different outcomes therefore favors data conditions that expose the discrepancy, providing targeted supervision for diagnostic construction. To select diagnostic test values, we apply the following execution-difference criterion:

$$\mathrm{Exec}(\sigma_{\text{issue}}; (S', V')) \neq \mathrm{Exec}(\sigma^*; (S', V')), \tag{2}$$

where σ^* denotes the reference solution and Exec denotes the query execution outcome. We retain construction trajectories whose test values satisfy this criterion, allowing the local models to learn from examples that expose SQL discrepancies. The reference solution provides an offline selection signal and is not an input to the deployed local constructors. The input and output definitions for the two supervised generation tasks are given in Section 3.4; model implementation and training compute are reported in Appendix D.2.

C.3 LOCAL CONSTRUCTION AND VALIDATION

At deployment, the schema model generates DDL from the issue description, problematic SQL, and original database schema. The data model then generates diagnostic values and DML conditioned on the constructed schema and issue context. The resulting environment is checked for construction executability, constraint compliance, and issue fidelity, as described in Section 3.3. If validation rejects a construction, the local variant restarts schema and data generation through fresh resampling,

without retaining the rejected artifacts or validation history as regeneration context. This restart policy differs from the LLM-based constructor's coupled reconstruction, which uses prior artifacts and diagnostics to guide revisions. Only validated environments are retained for subsequent debugging.

Teacher-model calls are used during offline training-data preparation. Locality refers to deployment-time environment construction; the downstream debugging agent may still use an external model, as in the reported local-construction experiments.

D Experiment Details

This appendix provides additional implementation details for the experimental protocol, including the execution settings, agentic baselines, repository-level evaluation workflow, and model configurations. Unless otherwise stated, Databox is used as a drop-in replacement for the real database: the agent workflow, prompt template, stopping rule, output format, and reasoning budget are kept unchanged, while the executable environment is switched from the original database E_{real} to the constructed Databox E'.

D.1 Estimated Execution Cost.

Wall-clock execution time is highly sensitive to hardware configuration, system load, caching behavior, and concurrent processes, making it difficult to reproduce and compare across different evaluation environments. To ensure stability and comparability, we adopt the optimizer-estimated query cost derived from the database query execution plan (QEP) as our execution efficiency metric. This cost reflects the planner's estimated CPU and I/O workload and provides a proxy for execution complexity that is less sensitive to runtime condition.

We focus on data query language (DQL) statements when computing execution cost. This choice is principled: optimizer-estimated costs are formally defined and consistently available for query statements via QEPs, whereas other SQL types (e.g., DDL or DML) do not yield comparable cost estimations across systems. Moreover, DQL statements constitute the dominant portion of real-world SQL workloads (88%–99.8%) and typically represent the primary execution bottleneck in database applications. We also observe that DQL queries dominate LLM agent exploration during iterative debugging (over 94% of executed SQL statements), indicating that improvements in DQL plan cost directly translate into reduced end-to-end debugging time. Therefore, average estimated DQL cost serves as a rigorous and representative proxy for overall execution efficiency.

To further enhance robustness, we report a 5% trimmed mean of the estimated costs, removing both the highest and lowest 5% of values. Extremely high-cost queries often correspond to cases that would result in execution timeouts under real execution, while extremely low-cost queries may arise from trivial or degenerate plans. Trimming mitigates the influence of such outliers and yields a more stable and fair comparison across methods.

End-to-end prompting. We include two execution-free baselines. **Simple Prompt** directly asks the model to repair the issued SQL according to the issue report and database context. **Chain-of-Thought** uses the same input but asks the model to reason through the issue before producing the final SQL. Neither baseline is allowed to execute SQL queries or observe database feedback. The final answer is parsed as the submitted SQL solution.

Agentic SQL baselines. We evaluate five execution-guided SQL baselines that share the same high-level interaction pattern: the agent reasons about the reported issue, executes SQL probes, observes results or error messages, and revises its candidate fix until it submits a final corrected SQL query or reaches the interaction budget. SQL-ACT is used as the SQL issue-solving-oriented baseline. Spider2Agent and CHASESQL are adapted from text-to-SQL agents by changing the objective from generating a SQL query from scratch to repairing the given issued SQL while preserving intent-consistent components of σ_{issue}. IntercodeSQL and AutoGPT-SQL are adapted from interactive code-environment or autonomous coding-agent scaffolds by exposing SQL execution as the main tool and rewriting the task prompt from query generation to issue repair. Across these baselines, we preserve the original agent workflow as much as possible and only make minimal task-specific changes to the prompt, tool description, and final-answer format required by SQL issue resolution.

Model	GPUs	Wall-clock hours	GPU hours
QWEN2.5-CODER-INSTRUCT	3×H100	26	78

Table 6: Training compute for Databox-Local. GPU hours are computed as the number of GPUs multiplied by wall-clock training hours.

File or directory	Role in the repository-level task
`README_TASK.md`	Entry-point instruction that describes the SQL debugging goal, available files, and required final output format.
`issue.md`	User-facing issue report, including the processed database id, SQL dialect metadata, natural-language issue description, and the issued SQL block.
`issue_sql.sql`	The buggy SQL artifact that the agent should diagnose and repair.
`db_schema.txt`	Preprocessed schema context for the target database, with large sample-row dumps removed to avoid exposing unnecessary data.
`db_guidelines.md`	Instructions for interacting with PostgreSQL through standard command-line tools such as `psql`.
`db_env.sh`	Environment script generated during deployment, exporting the host, port, user, password, and database name for the active execution environment.
`sandbox_intro.md`	Databox-only note describing the sandbox database and the initialization history used to construct it.
`artifacts/` and `logs/`	Writable directories for intermediate notes, execution traces, and run metadata.
`pred.json`	Mandatory final artifact containing the instance id and a list of corrected SQL strings.

Table 7: Per-instance repository layout used in BIRD-CRITIC-REPO.

D.2 MODEL IMPLEMENTATION DETAILS

For inference with backbone LLMs, we evaluate GEMINI-2.0-FLASH, CLAUDE-3.7-SONNET, and QWEN3-CODER-480B-A35B-INSTRUCT. All inference experiments use deterministic decoding with temperature set to 0, a maximum context length of 4096 tokens, and at most five reasoning-action iterations for SQL-level agentic baselines. Early stopping is allowed according to each agent's original termination criterion. GEMINI-2.0-FLASH is also used as the teacher model for schema-generation data in Databox-Local and for generating issue-report rewrites during QEP-driven training data construction.

For Databox-Local training, we use QWEN2.5-CODER-INSTRUCT as the base model for both the schema-generation and test-value-generation SLMs, trained with LoRA fine-tuning. The training is conducted on 3×H100 GPUs. Based on the current training record of approximately 26 wall-clock hours, the total training compute is $3 \times 26 = 78$ H100 GPU-hours.

E CONSTRUCTION OF BIRD-CRITIC-REPO

BIRD-CRITIC-REPO extends BIRD-CRITIC-PG from standalone SQL repair to repository-level SQL debugging. Instead of presenting the issue as a single prompt, each benchmark instance is materialized as an independent working directory that can be solved by a coding agent through normal file inspection, shell commands, and database command-line tools. This construction is designed to match vibe-coding workflows, where the agent must recover task context from project files, inspect or execute artifacts when needed, and write the final solution back to the repository.

Instance layout. For each SQL issue, we create a directory named by its instance id, e.g., `PostgreSQL_i`. The directory contains a task instruction file, the user issue, the issued SQL, schema context, database-access instructions, and output/log folders. The core files are summarized in Table 7.

Repository materialization. The repository is generated from the metadata and schema files of BIRD-CRITIC-PG. For each instance, the preparation script copies a template directory, fills placeholders such as the instance id and database name, writes `issue.md` from the original issue report, writes `issue_sql.sql` from the problematic SQL, and writes `db_schema.txt` from the preprocessed schema. For real-database runs, the sandbox-specific file is removed and the database guideline keeps the real-database safety notice. For Databox runs, the same repository layout is retained, but `sandbox_intro.md` is filled with the Databox initialization history and the guideline marks the environment as isolated. Thus, the task-facing repository remains nearly identical across settings, while the database endpoint is changed underneath.

CLI and file-system deployment. The deployment harness separates the task repository from the database control plane. Before a coding agent starts, a preparation manager allocates a per-instance PostgreSQL database and writes `db_env.sh` into the instance directory. In the real-database setting, the manager clones or restores the corresponding original PostgreSQL database into an instance-specific database and applies required preprocessing SQL. In the Databox setting, the manager creates an empty PostgreSQL database in an instance-specific tablespace and replays the Databox initialization SQL derived from the sandbox construction history. Both variants expose the same local connection interface, with PostgreSQL available on `127.0.0.1:55432` and the active database specified through environment variables. Consequently, coding agents interact with either environment using ordinary shell and file-system operations rather than a specialized SQL-debugging API.

Coding-agent harness. We provide tool-specific wrappers for OpenCode, QwenCode, and Kimi-Code, but keep the benchmark contract shared across tools. Each wrapper launches the agent inside the instance directory, points it to `README_TASK.md`, sources `db_env.sh` when needed, records stdout/stderr logs, applies a timeout and retry budget, and validates that `pred.json` exists with the required schema. Batch runners maintain a small preparation window so that only a bounded number of per-instance databases are live at the same time; after an instance finishes, the manager records the terminal status and cleans the database resources. Finally, prediction collection scripts aggregate the per-instance `pred.json` files into a JSONL file for downstream evaluation. This design allows BIRD-CRITIC-REPO to evaluate repository-level SQL debugging while preserving a controlled comparison between real-database interaction and Databox interaction.

F Limitations and Future Work

This work focuses on execution-guided SQL issue resolution with task-specific sandbox environments, evaluated primarily on PostgreSQL-based SQL debugging and repository-level settings. While the results demonstrate strong cost reduction and competitive task performance, future work can further broaden Databox to additional database systems, SQL dialects, enterprise deployment scenarios, and richer multi-step data-engineering workflows, including high-value data science tasks such as exploratory analysis, feature engineering, data validation, and pipeline debugging.

G Model Token Usage and API Cost

Experimental setup. We evaluate the model-side cost of Databox across SQL-ACT, IntercodeSQL, and Spider2Agent, using Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct as backbone models. For each agent–model combination, we compare real-database interaction with Databox interaction, including the cost of constructing Databox in the latter. We report aggregate input and output tokens and estimated API expenditure, with a separate decomposition into construction and agent reasoning. Here, agent reasoning includes all recovered prompt and response tokens during debugging, including environment descriptions and execution feedback. Simple Prompt and Chain-of-Thought serve as execution-free reference baselines.

Token accounting. The accounting includes only API calls for which both the prompt and response are available. To standardize token accounting and enable comparison across model providers, we use the `cl100k_base` encoding of `tiktoken` as a unified token-counting convention. The assumed

input/output prices, in USD per million tokens, are $0.15/0.60$ for Gemini-2.0-Flash, $3.00/15.00$ for Claude-3.7-Sonnet, and $1.50/7.50$ for Qwen3-Coder-480B-A35B-Instruct.

Overall comparison. Table 8 shows that Databox increases total token usage in all nine settings, by 17.1%–198.8% relative to Real DB. Although Databox substantially reduces database execution cost, estimated model API expenditure increases in eight of the nine settings. The largest relative increase occurs with IntercodeSQL and Claude-3.7-Sonnet: token usage rises from 2.92 million to 8.72 million, and estimated cost rises from \$14.74 to \$32.19. As observed in the trajectory analysis, IntercodeSQL's token overhead is partly attributable to repeated SQL execution and delayed emission of the `[Done]` termination signal, which prolongs debugging beyond the necessary interactions. SQL-ACT with Gemini-2.0-Flash exhibits a different pattern, with total tokens increasing from 6.60 million to 7.73 million while estimated expenditure decreases from \$1.73 to \$1.61. In this setting, additional Databox context increases input-token usage, while shorter model responses reduce output-token usage. Because output tokens are priced four times higher than input tokens for Gemini-2.0-Flash, the output-side savings outweigh the additional input-side expenditure. More concise execution feedback can also reduce the context carried into subsequent calls.

Sources of overhead. The phase decomposition in Table 9 identifies construction as the principal contributor to the increase in total tokens. Construction accounts for 21.5%–36.4% of Databox token usage and more than half of the net token increase relative to Real DB in every setting. Reasoning costs exhibit a less uniform pattern. For SQL-ACT with Gemini-2.0-Flash and IntercodeSQL with Qwen3-Coder-480B-A35B-Instruct, reasoning tokens decrease by 22.0% and 10.5%, respectively, although construction offsets these savings in total token usage.

Databox also introduces context describing the constructed environment. In the SQL-ACT implementation, this context includes construction SQL, inserted data, reproduction output, and prior sandbox history. Repeated inclusion of these descriptions increases input usage independently of the physical size of the database.

Databox reduces the scale of the SQL execution environment while introducing construction and contextualization costs. Consequently, database execution efficiency and API-token efficiency need not improve together.

H Real Execution Time Evaluation

Experimental setup. We further evaluate Databox using real client-observed SQL wall-clock time to complement the optimizer-estimated execution cost reported in the main experiments. Unlike QEP-estimated cost, which provides a hardware-independent proxy of execution complexity, wall-clock measurement captures the elapsed time experienced during actual database interaction. Measurements are conducted on a shared KVM virtual machine running Ubuntu 22.04.5 LTS. The guest system reports an Intel Xeon Silver 4416+ processor, with the experiment account restricted to two CPU equivalents and 20.5 GiB memory. PostgreSQL 14.20 stores database files on an NFSv4.2 filesystem. The client uses Python 3.10.19 with psycopg2 2.9.11, and execution time is measured using a monotonic high-resolution clock.

Timing protocol. The reported wall-clock time includes connection establishment, SQL execution, complete result retrieval, transaction finalization, and connection teardown. Statements are executed serially within each benchmark instance with a 120-second statement timeout. Each instance starts from a newly restored database state, while restoration time is excluded from Real DB execution time. To reduce the influence of individual slow executions, we report mean, median, and 5%-trimmed mean runtime.

Overall comparison. Table 11 reports the real execution time results. The observed runtime trend is consistent with the optimizer-cost analysis reported in the main paper and demonstrates that Databox substantially reduces SQL replay overhead under the controlled execution environment. For example, with SQL-ACT and Gemini-2.0-Flash, the 5%-trimmed mean execution time decreases from 0.0545 seconds on Real DB to 0.0091 seconds on Databox, indicating that the execution-efficiency advantage observed from optimizer estimates also appears in measured elapsed execution time.

Table 8: Comparison of model token usage and estimated API cost for Real DB and Databox. Databox totals include both construction and agent reasoning. Δ Cost is the absolute change in USD, calculated as Databox minus Real DB using the displayed monetary values. Gemini, Claude, and Qwen denote Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct, respectively.

Method	Model	Environment	Total tokens	Cost (USD)	Δ Cost (USD)
SQL-ACT	Gemini	Real DB	6,598,903	1.73	
		Databox	7,725,128	1.61	-0.12
	Claude	Real DB	6,998,532	27.21	
		Databox	11,316,904	41.78	+14.57
	Qwen	Real DB	6,745,467	12.98	
		Databox	10,193,268	18.64	+5.66
IntercodeSQL	Gemini	Real DB	7,197,060	1.24	
		Databox	11,597,795	1.91	+0.67
	Claude	Real DB	2,919,417	14.74	
		Databox	8,722,257	32.19	+17.45
	Qwen	Real DB	4,635,296	8.97	
		Databox	6,403,601	11.15	+2.18
Spider2Agent	Gemini	Real DB	9,335,322	2.03	
		Databox	15,155,011	2.83	+0.80
	Claude	Real DB	8,422,648	31.33	
		Databox	14,118,415	49.62	+18.29
	Qwen	Real DB	6,673,166	12.61	
		Databox	11,671,358	20.11	+7.50

Table 9: Decomposition of Databox model usage into agent reasoning and sandbox construction. Each phase includes input and output tokens. Costs are in USD. The phase costs sum to the Databox totals in Table 8, using the reported two-decimal precision. Gemini, Claude, and Qwen denote Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct, respectively.

		Agent reasoning		Construction	
Method	Model	Tokens	Cost (USD)	Tokens	Cost (USD)
SQL-ACT	Gemini	5,148,394	1.22	2,576,734	0.39
	Claude	7,528,418	30.42	3,788,486	11.36
	Qwen	6,883,644	13.68	3,309,624	4.96
IntercodeSQL	Gemini	7,716,697	1.33	3,881,098	0.58
	Claude	5,547,555	22.67	3,174,702	9.52
	Qwen	4,148,521	7.77	2,255,080	3.38
Spider2Agent	Gemini	11,686,972	2.31	3,468,039	0.52
	Claude	10,798,986	39.66	3,319,429	9.96
	Qwen	9,163,043	16.35	2,508,315	3.76

Runtime distribution analysis. Wall-clock execution exhibits a heavy-tailed distribution for several agent–model combinations. For example, IntercodeSQL with Gemini-2.0-Flash on Real DB has a mean runtime of 5.83 seconds but a 5%-trimmed mean of 0.248 seconds, indicating that a small number of expensive executions dominate the average. Similar gaps appear for Databox settings, such as Spider2Agent with Gemini-2.0-Flash and IntercodeSQL with Qwen3-Coder-480B-A35B-Instruct. The trimmed statistic provides a more stable comparison by reducing the influence of extreme executions.

Table 10: Model token usage and estimated API cost of execution-free prompting baselines, measured under the same accounting protocol. Gemini, Claude, and Qwen denote Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct, respectively.

Method	Model	Total tokens	Cost (USD)
Simple Prompt	Gemini	839,608	0.16
	Claude	1,573,877	6.83
	Qwen	1,565,686	3.36
Chain-of-Thought	Gemini	1,860,453	0.38
	Claude	1,634,083	7.44
	Qwen	1,677,124	4.04

Table 11: Real execution time evaluation on the controlled PostgreSQL environment. Runtime is measured as client-observed wall-clock time in seconds, including connection establishment, SQL execution, result retrieval, transaction finalization, and connection teardown. Mean, median, and 5%-trimmed mean are reported. Gemini, Claude, and Qwen denote Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct, respectively.

Environment	Method	Model	SQL Ops	Failure Rate	Mean	Median	5%-Trim Mean
Real DB	SQL-ACT	Gemini	1759	34.11%	1.643680	0.031264	0.054495
		Qwen	2330	28.19%	0.675776	0.037776	0.051531
		Sonnet	2618	25.13%	0.308074	0.043199	0.053617
	IntercodeSQL	Gemini	2693	38.73%	5.830670	0.038420	0.247928
		Qwen	1858	36.72%	0.904230	0.027846	0.054741
		Sonnet	1285	30.00%	1.765361	0.018930	0.023959
	Spider2Agent	Gemini	1452	42.36%	1.093650	0.020652	0.043900
		Qwen	1239	37.89%	0.352882	0.019341	0.030740
		Sonnet	1476	38.00%	0.346630	0.022291	0.033316
Databox	SQL-ACT	Gemini	1103	20.04%	0.237526	0.006675	0.009147
		Qwen	2192	10.63%	0.249314	0.018390	0.018019
		Sonnet	2590	17.84%	0.022162	0.019995	0.020624
	IntercodeSQL	Gemini	2881	15.72%	0.022722	0.019873	0.019786
		Qwen	1601	15.59%	0.245141	0.010694	0.011476
		Sonnet	2759	21.96%	0.021886	0.020293	0.019660
	Spider2Agent	Gemini	1193	17.80%	1.189706	0.006663	0.008854
		Qwen	1313	12.29%	0.286970	0.008784	0.009646
		Sonnet	2009	11.19%	0.253050	0.017665	0.019302

I Synthetic Data Generation Strategy Ablation

Experimental setup. We investigate how different data generation strategies affect the effectiveness of Databox construction for SQL debugging. Unlike conventional synthetic data generation, which primarily aims to approximate the original data distribution or generate statistically plausible database contents, Databox generates issue-specific diagnostic values designed to expose the execution discrepancy underlying a reported SQL issue. This experiment compares Databox with alternative population strategies, including real-data sampling, preview-based construction, and general-purpose synthetic database generation. Table 12 summarizes the compared strategies.

We evaluate all strategies on BIRD-CRITIC-PG with SQL-ACT using Gemini-2.5-Flash as the backbone model. To isolate the effect of data generation, all methods operate on the same issue-specific reduced schema produced by the Databox construction pipeline, while only the data strategy used for Databox construction is varied. The downstream SQL debugging agent, interaction protocol, and evaluation procedure remain unchanged across all settings.

For preview-based methods, the provided source data consists of the first three rows of each table. For DOMINO (SchemaAnalyst) (Alsharif et al., 2018), we use the released DOMINO `dominoRandom` generator in SchemaAnalyst with the Integrity Constraint Coverage (ICC) criterion, then load its

constraint-valid fixture rows into the shared reduced Databox schema. For learned synthetic generation methods, we evaluate the Hierarchical Modeling Algorithm (HMA) synthesizer from SDV (Patki et al., 2016) and CTGAN (Xu et al., 2019). These methods access source data locally during model fitting and generate synthetic rows that are subsequently used to populate the reduced schema.

Evaluation metrics. We report two success-based metrics to evaluate each data generation strategy. The primary success rate (Primary SR) is computed as the percentage of all 505 evaluated instances for which the final SQL solution passes the official evaluation. This metric captures the complete pipeline effectiveness, including failures during Databox construction. We additionally report the conditional success rate (Conditional SR), which computes the success rate only among successfully constructed sandbox databases. This metric separates downstream debugging performance from construction failures.

Overall comparison. Table 4 summarizes the results of different data generation strategies. Databox without preview achieves the highest end-to-end success rate, obtaining 190 successful cases out of 505 instances (37.62%). Preview-row copy and Bernoulli sampling achieve comparable but lower success rates, with 180 (35.64%) and 179 (35.45%) successful cases, respectively. These results indicate that directly exposing or sampling real data does not necessarily provide the diagnostic conditions required for SQL issue resolution.

General-purpose synthetic generation methods also construct executable database instances but achieve lower end-to-end success rates. DOMINO (SchemaAnalyst) constructs 420 out of 505 sandboxes and achieves 149 successful cases (29.50%). SDV-HMA and CTGAN achieve lower primary success rates of 14.65% and 26.14%, respectively, partly due to construction failures under the reduced schema setting. Their conditional success rates among successfully constructed cases are closer to other approaches, suggesting that the primary performance gap mainly comes from the ability to construct applicable debugging environments.

Issue-specific generation versus general synthesis. The results demonstrate that generating valid database contents is not equivalent to constructing an effective debugging environment. General-purpose generators such as DOMINO, SDV-HMA, and CTGAN focus on producing structurally valid or statistically plausible database instances, while Databox targets the specific execution conditions needed to reveal SQL issues. For SQL debugging, generated values must distinguish incorrect and correct SQL behaviors rather than merely resemble the original database distribution. The higher end-to-end success rate of Databox suggests that issue-guided data generation better matches the requirements of execution-based debugging in this setting.

Effect of preview data. Providing additional source data does not improve Databox performance in this setting. The preview-based configurations use the first three rows of each table and are not optimized for reproducing specific SQL issues. Preview rows can bias the construction process toward directly copying observed values rather than generating issue-specific diagnostic cases. Moreover, copied rows may introduce compatibility problems with the reduced schema, including missing relational structures or constraints required by the debugging task. Consequently, exposing more real examples does not necessarily produce more informative execution feedback for SQL debugging.

Data exposure considerations. Different data generation strategies provide different levels of source-data access depending on their construction process. Sampling-based approaches directly reuse source records and therefore require access-control considerations when real values are exposed to external models. Learned synthetic generation methods such as SDV-HMA and CTGAN instead fit generation models using source data locally and expose generated samples during downstream interaction, but the fitting process still requires access to the original data and does not provide formal differential privacy guarantees by default. In contrast, Databox generates issue-specific diagnostic values from schema and debugging context without requiring access to source records. These differences highlight the practical trade-off between source-data exposure and debugging effectiveness when constructing executable sandbox environments.

Table 12: Comparison of data generation strategies evaluated in the ablation study. All methods operate on the same issue-specific reduced Databox schema; the comparison varies only the data strategy used for Databox construction. FK denotes foreign key, and ICC denotes Integrity Constraint Coverage.

Method	Data generation strategy
Databox, no preview	Issue-guided diagnostic value generation using the issue description, SQL query, and reduced schema without accessing source data rows.
Databox + preview	Issue-guided diagnostic value generation additionally conditioned on the first three rows of each table from the original database.
Preview-row copy	Directly transfers the first three source rows per table into the reduced schema, with compatibility processing for schema constraints.
Bernoulli sample + FK closure	Samples real database rows using Bernoulli sampling and adds required foreign-key closure rows to construct a valid reduced-schema database.
DOMINO (SchemaAnalyst)	Uses the released DOMINO `dominoRandom` generator with the Integrity Constraint Coverage (ICC) criterion to generate constraint-valid fixture rows, which are then loaded into the shared reduced Databox schema.
SDV-HMA	Fits SDV's Hierarchical Modeling Algorithm (HMA) synthesizer on locally available source data and generates synthetic rows that are subsequently used to populate the reduced schema.
CTGAN + relational adapter	Generates synthetic table values using CTGAN and applies a relational adapter for key assignment, foreign-key mapping, and constraint validation.

J Agent Action Effort

The efficiency of execution-guided debugging depends on both the cost of individual SQL queries and the number of agent interactions required to produce a repair. Repeated schema inspection, hypothesis testing, and candidate revision accumulate costs across the debugging trajectory. We compare action effort under RealDB and Databox in agent-level and repository-level experiments to examine whether Databox also reduces the number of interactions, complementing the per-query execution-cost analysis.

Metrics. For agent-level experiments, we report the *average number of action commands per instance*, where each emitted command constitutes one action turn. This count includes SQL execution, inspection, reset, termination, and malformed action attempts. For repository-level experiments, we extract two measures of interaction effort from agent logs. The *parsed command count* includes command-like shell, file, and database operations, while the *parsed PostgreSQL execution count* counts SQL statements recovered from PostgreSQL command invocations. The latter counts recovered statements regardless of execution success. Each metric is averaged over trajectories within the corresponding method–model–environment setting.

J.1 Agent Action Taxonomy

For the action-distribution analysis in Figure 6, we exclude special termination and completion-flag actions and categorize the remaining actions into five types. *Correction* denotes actions where the agent directly tests a candidate solution. *Inspection* denotes actions where the agent examines tables, schema information, or previewed data. *Database modification* denotes actions where the agent changes the database schema or modifies data. *Partial SQL solution* denotes actions where the agent intentionally solves part of the task and observes the resulting state change. *Validation* denotes actions where the agent writes SQL queries to verify prior execution results or database state.

Experimental setup. We use Gemini-2.0-Flash, Claude-3.7-Sonnet, and Qwen3-Coder-480B-A35B-Instruct, consistent with the main experiments. Agent-level runs allow up to five reasoning–action iterations, with scaffold-specific early stopping. Repository-level runs use each coding tool's stopping criterion without an explicit action-turn cap, subject to the harness timeout and retry budgets.

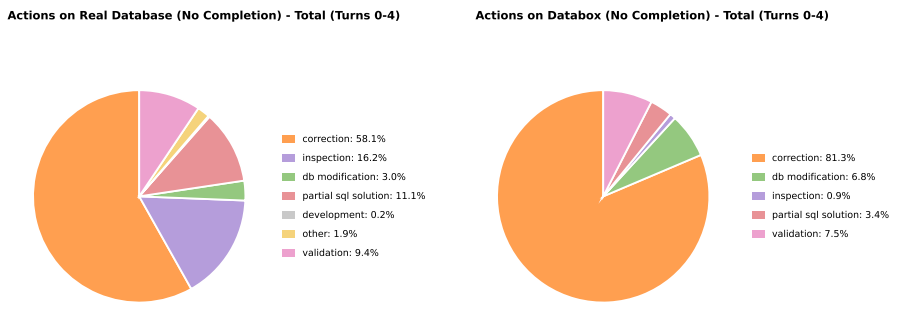


Figure 6: Agent action distribution on Databox and RealDB, excluding special termination and completion-flag actions. Action-type definitions are provided in Appendix J.1.

Table 13: Average number of action commands per instance in agent-level experiments. $\Delta = \text{Databox} - \text{RealDB}$; percentage change is $100\Delta/\text{RealDB}$. Negative values indicate reductions.

Method	Model	RealDB	Databox	Δ	Change (%)
SQL-ACT	Gemini-2.0-Flash	3.957	2.925	-1.032	-26.1
SQL-ACT	Qwen3-Coder-480B-A35B-Instruct	4.664	4.725	$+0.061$	$+1.3$
SQL-ACT	Claude-3.7-Sonnet	4.847	4.655	-0.192	-4.0
IntercodeSQL	Gemini-2.0-Flash	4.915	4.783	-0.132	-2.7
IntercodeSQL	Qwen3-Coder-480B-A35B-Instruct	3.925	3.542	-0.383	-9.8
IntercodeSQL	Claude-3.7-Sonnet	2.506	4.375	$+1.869$	$+74.6$
Spider2Agent	Gemini-2.0-Flash	4.598	3.294	-1.304	-28.4
Spider2Agent	Qwen3-Coder-480B-A35B-Instruct	4.411	4.191	-0.220	-5.0
Spider2Agent	Claude-3.7-Sonnet	4.834	4.623	-0.211	-4.4

Agent-level results. Databox generally reduces the average number of action turns, with decreases in seven of the nine method–model settings (Table 13). The largest reductions occur with GEMINI-2.0-FLASH. SQL-ACT decreases from 3.957 to 2.925 actions per instance (26.1%), and Spider2Agent decreases from 4.598 to 3.294 (28.4%). IntercodeSQL with QWEN3-CODER-480B-A35B-INSTRUCT also decreases from 3.925 to 3.542 actions (9.8%), showing that the reduction extends across model backbones. There are two exceptions. SQL-ACT with QWEN3-CODER-480B-A35B-INSTRUCT increases slightly from 4.664 to 4.725 actions (1.3%), whereas IntercodeSQL with CLAUDE-3.7-SONNET increases substantially from 2.506 to 4.375 (74.6%). Notably, fewer actions alone do not establish more efficient successful debugging. Short trajectories may reflect premature termination or failure, while longer trajectories may involve additional productive validation.

Termination and repetition in IntercodeSQL. The increase with CLAUDE-3.7-SONNET primarily reflects the interaction protocol. Within the five-iteration budget, IntercodeSQL terminates early only when the model explicitly emits `[Done]`; successful execution of a candidate SQL statement does not trigger termination. Trajectory analysis shows that the model often generates a new response that resolves to the same SQL action without signaling completion. Collapsing consecutive identical actions reduces the mean command counts to approximately 2.44 for RealDB and 3.33 for Databox, narrowing the gap from 1.869 to approximately 0.89 commands per instance. Thus, excess consecutive repetition accounts for roughly half of the original difference. Databox's encouragement of unrestricted sandbox experimentation and inclusion of the construction transcript, combined with the continuation instruction to *provide a corrected SQL command*, favor another SQL test over an explicit completion signal. Termination and repetition artifacts therefore account for much of the increased turn count. This repetition exposes a flaw in our experimental design. The interaction protocol did not suppress consecutive identical SQL actions when the execution state was unchanged, allowing redundant executions that inflated action counts and incurred unnecessary execution costs.

Repository-level results. Databox consistently reduces repository-level interaction effort across all three model backbones (Table 14). Mean parsed command counts decrease from 14.3 to 10.3

Table 14: Average numbers of parsed commands and PostgreSQL statements per trajectory in repository-level experiments. $\Delta = \text{Databox} - \text{RealDB}$; percentage change is $100\Delta/\text{RealDB}$.

Model	RealDB	Databox	Δ	Change (%)
Parsed commands				
Qwen3-Coder-480B-A35B-Instruct	14.3	10.3	-4.0	-28.0
Claude-3.7-Sonnet	11.6	8.5	-3.1	-26.7
Gemini-2.0-Flash	15.1	10.9	-4.2	-27.8
Parsed PostgreSQL statements				
Qwen3-Coder-480B-A35B-Instruct	7.9	5.9	-2.0	-25.3
Claude-3.7-Sonnet	8.8	6.8	-2.0	-22.7
Gemini-2.0-Flash	7.8	5.8	-2.0	-25.6

Table 15: Average numbers of parsed commands and PostgreSQL statements per trajectory by coding tool with Gemini-2.0-Flash. $\Delta = \text{Databox} - \text{RealDB}$; percentage change is $100\Delta/\text{RealDB}$.

	Commands				PostgreSQL statements			
Tool	RealDB	Databox	Δ	Change (%)	RealDB	Databox	Δ	Change (%)
KimiCode	18.5	13.7	-4.8	-25.9	7.1	5.8	-1.3	-18.3
OpenCode	6.8	5.7	-1.1	-16.2	6.4	4.9	-1.5	-23.4
QwenCode	20.1	13.2	-6.9	-34.3	9.8	6.9	-2.9	-29.6

for QWEN3-CODER-480B-A35B-INSTRUCT (28.0%), from 11.6 to 8.5 for CLAUDE-3.7-SONNET (26.7%), and from 15.1 to 10.9 for GEMINI-2.0-FLASH (27.8%). Mean parsed PostgreSQL execution counts decrease by 2.0 statements per trajectory for each model, corresponding to reductions of 22.7%–25.6%. Unlike the agent-level experiments, which enforce a five-iteration limit, the repository-level experiments allow each coding tool to determine when to stop. Interaction lengths can therefore extend beyond five rounds, exposing differences that a fixed turn budget can truncate. The more pronounced reductions in this setting reflect shorter debugging trajectories under tool-determined stopping, as measured by the command and PostgreSQL execution counts.

Differences across coding tools. With GEMINI-2.0-FLASH on the same repository tasks, the three tools exhibit distinct interaction lengths (Table 15). Under RealDB, OpenCode averages 6.8 commands per trajectory, compared with 18.5 for KimiCode and 20.1 for QwenCode. The corresponding PostgreSQL statement counts are 6.4, 7.1, and 9.8, respectively; thus, the variation in overall command counts is considerably larger than the variation in database interactions. This pattern reflects differences in how the tools organize repository inspection and debugging around database execution. Databox reduces command counts for all three tools, to 5.7, 13.7, and 13.2, respectively. QwenCode exhibits the largest reduction, with 6.9 fewer commands (34.3%) and 2.9 fewer PostgreSQL statements (29.6%) per trajectory. KimiCode also shows a substantial command reduction of 4.8 (25.9%), while OpenCode's shorter trajectories decrease by 1.1 commands (16.2%).

Focused debugging and interaction efficiency. The reduction in action turns reinforces the observed shift in action distributions. Agents using RealDB devote a greater share of their actions to schema inspection and partial or indirect SQL debugging, whereas Databox promotes more direct correction. By restricting the schema to task-relevant components and providing diagnostic test values, Databox reduces the need for repeated context gathering and exploratory queries. The shorter trajectories complement this behavioral shift, supporting the conclusion that Databox focuses the debugging process and helps agents converge with fewer interaction rounds. Databox thus reduces interaction overhead through two complementary mechanisms. Simplifying the database environment lowers the execution cost of individual SQL queries, while promoting more decisive agent actions reduces the cumulative cost of repeated interaction turns.

Table 16: Execution feedback length under a common formatter. Means are measured in tokens per event. The trimmed mean excludes the longest 5% of feedback events within each condition.

Model	Setting	Events	Direct mean	Trimmed mean
	SQL-ACT			
Gemini-2.0-Flash	RealDB	2,097	176.92	150.15
	Databox	1,550	75.29	53.69
Qwen3-Coder-480B-A35B-Instruct	RealDB	2,472	132.66	107.40
	Databox	2,504	89.54	70.61
Claude-3.7-Sonnet	RealDB	2,569	158.11	134.23
	Databox	2,467	108.10	85.02
	IntercodeSQL			
Gemini-2.0-Flash	RealDB	2,605	235.69	211.00
	Databox	2,535	81.41	61.86
Qwen3-Coder-480B-A35B-Instruct	RealDB	2,080	191.38	165.63
	Databox	1,877	93.48	71.57
Claude-3.7-Sonnet	RealDB	1,328	119.76	95.32
	Databox	2,319	112.70	91.03
	Spider2Agent			
Gemini-2.0-Flash	RealDB	2,437	165.13	138.29
	Databox	1,746	50.66	34.56
Qwen3-Coder-480B-A35B-Instruct	RealDB	2,338	128.93	105.41
	Databox	2,221	90.94	74.05
Claude-3.7-Sonnet	RealDB	2,562	165.19	142.58
	Databox	2,450	115.05	93.60

K Execution Feedback Length

Metrics. To examine how Databox affects the context introduced by execution feedback, we measure the number of tokens per feedback event using the `cl100k_base` tokenizer. Feedback is represented with a common formatter containing execution status and the result, error, or control message. We report the number of recovered feedback events, the direct mean over all events, and an upper-5%-trimmed mean that discards the longest $\lceil 0.05N \rceil$ events within each condition, where N is the event count.

Results. Databox produces shorter execution feedback in all nine method–model settings under both averaging schemes (Table 16). Direct mean reductions range from 5.9% to 69.3%. With Gemini-2.0-Flash, SQL-ACT decreases from 176.92 to 75.29 tokens per event (57.4%), IntercodeSQL from 235.69 to 81.41 (65.5%), and Spider2Agent from 165.13 to 50.66 (69.3%). Reductions persist after upper-tail trimming, showing that the effect extends beyond exceptionally long outputs. These results show that Databox can alleviate the long-context burden by reducing the amount of execution feedback added to the agent's context.

L Database Mutation During Repository-Level Debugging

Setup and measurement. We examine database mutation during repository-level debugging on 100 task IDs using Qwen3-Coder-480B-A35B-Instruct with three vibe coding tools, KimiCode, OpenCode, and QwenCode, yielding 300 trajectories per environment. To screen for potential database changes without clear user intent, we flag trajectories containing apparent unauthorized mutating SQL during exploration, including `INSERT`, `UPDATE`, `DELETE`, `ALTER`, `CREATE`, `DROP`, and trigger, function, or extension creation. We report the number of flagged trajectories and the total number of unintended mutating SQL mentions.

Mutation frequency and safety implications. Apparent mutating SQL occurs in 54 of 300 RealDB trajectories (18.0%), comprising 174 SQL mentions (Table 17). The frequency varies

Table 17: Apparent database mutation in the 100-task repository sample. Each environment includes 300 trajectories across three vibe coding tools, with 100 trajectories per vibe coding tool and environment.

Setting	Flagged	Total	SQL mentions	Rate (%)
RealDB	54	300	174	18.0
Databox	59	300	198	19.7

	RealDB		Databox	
Vibe coding tool	Flagged	Rate (%)	Flagged	Rate (%)
KimiCode	1	1.0	4	4.0
OpenCode	22	22.0	24	24.0
QwenCode	31	31.0	31	31.0

Table 18: Final correctness conditional on the presence of apparent mutating SQL.

Setting	Mutating SQL	Correct	Total	Success rate (%)
RealDB	Not detected	101	246	41.1
RealDB	Detected	32	54	59.3
Databox	Not detected	83	241	34.4
Databox	Detected	34	59	57.6

substantially across vibe coding tools, from 1.0% for KimiCode to 22.0% for OpenCode and 31.0% for QwenCode. Databox exhibits a slightly higher overall frequency of 19.7%, with 198 SQL mentions. These frequencies show that exploratory database mutation is not uncommon in repository-level agent debugging. Under RealDB, the same exploratory behavior targets the real database and can introduce data or schema changes beyond the submitted repair, potentially contaminating database contents, corrupting records, or causing data loss. Databox contains this risk by isolating diagnostic mutations within a disposable, task-specific sandbox, without requiring agents to avoid all database modifications.

Relationship with correctness. Trajectories containing apparent mutating SQL have higher success rates in both environments (Table 18). Under RealDB, the success rate is 59.3% for flagged trajectories and 41.1% for unflagged trajectories, a difference of 18.2 percentage points. The corresponding rates under Databox are 57.6% and 34.4%, a difference of 23.2 percentage points. This positive association is consistent with database modification serving a diagnostic role, such as constructing test conditions or checking hypotheses. Notably, 32 of the 54 flagged RealDB trajectories produce correct final solutions, demonstrating that mutation-risk behavior also occurs in successful runs. Benchmark evaluations based solely on accuracy or success rate can therefore overlook unintended database changes during debugging. Evaluating these side effects separately from final-answer correctness is necessary to expose this failure mode.

M SPIDER2-LITE SQLITE GENERALIZATION STUDY

Experimental setup. We further evaluate whether Databox can serve as a compact substitute for the original SQLite databases on 106 Spider2-Lite text-to-SQL instances. Unlike the SQL debugging setting, Spider2-Lite provides a natural-language question rather than an issued SQL query and issue report, and it evaluates SQLite execution rather than PostgreSQL execution. Because there is no reported SQL issue to reproduce, Databox construction generates data directly from the question and schema context without applying the issue-guideline stage. For each instance, we run paired Databox and Real DB conditions with three agent scaffolds: Spider-Agent, IntercodeSQL, and SQL Act. Each paired run permits at most five model calls, three API retries, and a bounded process timeout, yielding 636 trajectories per model configuration.

Table 19: Final-query median execution time on the 106-instance Spider2-Lite SQLite. Speedup is Real DB median time divided by Databox median time.

Model	Agent	Databox median (ms)	Real DB median (ms)	Speedup
		Qwen3-Coder		
Qwen3-Coder	Spider-Agent	2.159	41.071	19.02×
Qwen3-Coder	IntercodeSQL	1.786	26.863	15.04×
Qwen3-Coder	SQL Act	1.749	19.791	11.32×
		Gemini-2.5-Flash		
Gemini-2.5-Flash	Spider-Agent	3.045	32.115	10.55×
Gemini-2.5-Flash	IntercodeSQL	1.899	40.357	21.25×
Gemini-2.5-Flash	SQL Act	1.870	38.561	20.63×

Table 20: Execution pass rates on the 106 instances from Spider2-Lite SQLite. Δ denotes Databox minus Real DB in percentage points.

Model	Agent	Databox	Real DB	Δ
		Qwen3-Coder		
Qwen3-Coder	Spider-Agent	35/106 (33.02%)	39/106 (36.79%)	−3.77
Qwen3-Coder	IntercodeSQL	35/106 (33.02%)	35/106 (33.02%)	0.00
Qwen3-Coder	SQL Act	35/106 (33.02%)	34/106 (32.08%)	+0.94
		Gemini-2.5-Flash		
Gemini-2.5-Flash	Spider-Agent	46/106 (43.40%)	42/106 (39.62%)	+3.77
Gemini-2.5-Flash	IntercodeSQL	47/106 (44.34%)	53/106 (50.00%)	−5.66
Gemini-2.5-Flash	SQL Act	45/106 (42.45%)	51/106 (48.11%)	−5.66

Models and metrics. We evaluate two backbone models, QWEN3-CODER and GEMINI-2.5-FLASH. We report both count-level accuracy and percentage accuracy over 106 instances. Following the execution-cost analysis in the main experiments, we also report median final-query execution time to characterize execution efficiency in the SQLite setting.

Execution efficiency. Table 19 shows that Databox still substantially reduces final-query execution time when generalized to text-to-SQL tasks and the SQLite dialect. Median final-query speedups range from $10.55\times$ to $21.25\times$ for Gemini-2.5-Flash and from $11.32\times$ to $19.02\times$ for Qwen3-Coder.

Accuracy preservation. Table 20 shows that Databox maintains competitive pass rates on Spider2-Lite, with accuracy varying across backbone models and agent scaffolds. Across the evaluated settings, Databox differs from Real DB by at most 5.66 percentage points and does not introduce a consistent performance decline. These results suggest that Databox generalizes beyond PostgreSQL-style SQL issue repair. However, the Spider2-Lite setting exposes additional requirements for text-to-SQL sandbox construction. Compared with SQL debugging tasks, the text-to-SQL setting provides weaker construction signals because it contains only a natural-language question, without a problematic SQL query or explicit issue description. As a result, the data generation stage cannot directly apply issue-guided reproduction guidelines and must rely on less targeted question- and schema-conditioned synthetic data.

N PROMPT TEMPLATES

This appendix lists the prompt templates used in the implementation. Placeholders such as `{schema}`, `{user_issue}`, and `{issue_sql}` are filled with the corresponding instance-specific content before inference. This prompt template includes the three-line preview of the real data view for each table, while no-preview setting is used simply by removing the preview lines.

N.1 DATABOX CONSTRUCTION PROMPT

The following prompt is used for the Databox sandbox-building phase. The model interacts with an initially empty PostgreSQL sandbox and is asked to construct a minimal reproduction environment for the reported SQL issue, without producing the final corrected SQL.

```
Interact with an initially empty PostgreSQL sandbox database. Your task
is to
build a reproduction environment for the user's SQL issue by
constructing a
reproduced schema, inserting data, and finally reproducing the bug
reported by
the user.

You will be given the following information:

1. Database schema and real data view: the user's actual CREATE TABLE
DDL plus
a small "First 3 rows" preview for each table. These preview rows come
from the
real database. They are not exhaustive, but they show realistic values,
formats,
null patterns, relationships, and edge cases.

2. User Issue: a natural language description of the desired outcome or
current
bug.

3. Problematic SQL: the query (or queries) that presently fails to meet
the
requirement.

Your goal in this phase is only to build a sandbox that can reproduce
the issue:
- Recreate the necessary schema in the sandbox database.
- Insert minimal data required to trigger the issue.
- Prefer using, adapting, or extending the real data view rows when they
are
  relevant. You may copy values from the shown rows into INSERT
  statements if
  they help construct realistic test data.
- Add synthetic rows only when the shown real data rows are insufficient
to
  reproduce the issue.
- Execute the exact problematic SQL to confirm the issue can be
reproduced in
  the sandbox database.
- Output [Done] in action when the issue has been successfully
reproduced. Do
  not diagnose the root cause or produce the corrected SQL in this
  sandbox-building phase. If the empty database is sufficient to
  reproduce the
  issue, you can directly output [Done] without any prior actions.

Use interleaving Thought, Action, Observation steps.
```

```
Thought can reason about the process of building reproduced database,
such as:
- Which tables, columns, constraints, and relationships are needed to
recreate
  schema.
- Which real data view rows are useful as seed examples.
- Whether additional synthetic rows are needed to trigger the issue.
- Whether the observed result sufficiently demonstrates the problem.

Action must be executable PostgreSQL statements (except for
ResetSandbox, which
is a special non-SQL action), and must be one of the following:

- Schema Creation: PostgreSQL that create or modify tables, types,
functions or
  constraints (e.g., `CREATE TABLE`, `ALTER TABLE`, etc.).

- Data Population: PostgreSQL that insert test data in tables to
reproduce the
  issue (e.g., `INSERT` statements).

- Schema Inspection: PostgreSQL that reveal table / column information
  (e.g., querying `information_schema`).

- Sample Data: PostgreSQL that preview example rows of a table
  (e.g., `SELECT * FROM table_name LIMIT 3`).

- Bug Reproduction: PostgreSQL that execute the problematic SQL on the
  reproduced database to confirm the issue in the sandbox.

- ResetSandbox: A special non-SQL action that instructs the system to
discard
  the current sandbox database and initialize a new empty one. This
  action
  should be invoked by directly placing `ResetSandbox` inside the
  `<action>`
  tag. (e.g. <action>ResetSandbox</action>) At the beginning, the empty
  sandbox
  database has already been initialized for you. You should only use
  ResetSandbox if you have made changes that require a fresh start.

The observation will be the execution results from the sandbox
environment.

Wrap your thought in `<thought>[Your Thought]</thought>` and your action
in
`<action>[Executable SQL or ResetSandbox]</action>`.

The input is as follows:

### Database Schema And Real Data View
{schema}

### User Issue
{user_issue}

### Problematic SQL
{issue_sql}

MOST IMPORTANT:
- Your output must contain both thought and action parts.
- Wrap your thought in the `<thought>[Your Thought]</thought>` tag.
- Wrap your action in `<action>[Executable SQL or
ResetSandbox]</action>`.
- The action inside the `<action></action>` tags must be pure PostgreSQL
```

```
  statements (or exactly `ResetSandbox`) that can be executed directly,
  without
  any comments or post-processing.
- Only generate one round of thought and action. Generating multiple
rounds at
  once is NOT ALLOWED!
- In one action turn, generate multiple related SQL statements together
when
  useful.
- At the beginning, the empty sandbox database has already been
initialized, so
  do not use ResetSandbox in the first round unless the first action
  truly needs
  a fresh restart.
- Do not solve or correct the issue SQL in this phase. Build and
validate the
  reproduction sandbox only.

Only generating the next round of thought and action. Generating
multiple rounds
at once is NOT ALLOWED!

Now generate the first thought and action for the input.

### React
<thought>
```

N.2 Repository-level Vibe-coding Prompt

The following prompt is used as the entry-point task instruction in each BIRD-CRITIC-REPO instance. It is placed in `README_TASK.md` and read by the coding-agent harness at the start of a repository-level run. In the real-database variant, the `sandbox_intro.md` item is removed from the file list; in the Databox variant, it is retained and filled with the sandbox initialization history.

```
# Task: Fix the SQL

Working on the database `{db_name}`, and solve the user's issue:

You are given:
- `issue.md`: user description of their issue and what they want to
achieve
  with SQL
- `issue_sql.sql`: buggy SQL to fix (may contain multiple SQL statements)
- `db_schema.txt`: database schema + sample rows
- `db_guidelines.md`: how to interact with PostgreSQL database
- `sandbox_intro.md`: information about the sandbox database and its
  initialization history

## Goal
Produce corrected SQL that solves the issue and satisfies the user
request.

You may use bash commands, file operations, and python code to help you
understand the issue and explore the database. Finally you should write
the
corrected SQL in `pred.json` as described below.

## Output format (MANDATORY)
Write a file named `pred.json` in this folder with **exactly** this JSON:

```json
{
  "instance_id": "{{instance_id}}",
  "predicted_sql": ["<SQL1>", "<SQL2>", "..."]
```
```

```
}
```

```
Rules:

- Always explore the working directory and check the files before you
write the
  `pred.json`. The files may contain useful information that can help you
  understand the issue and fix the SQL.

- `predicted_sql` must always be a JSON list (even if there is only one
SQL).

- Put SQL strings only. No markdown fences or comments.

- Before you write the predicted SQL and finish, make sure to test the
SQL in
  the database environment. Detailed instructions on how to do that are
  in
  `db_guidelines.md`. You can also use bash commands and python code to
  help
  you test the SQL.

- Besides testing, interact with the database can also help you
understand the
  issue and explore the error. You are encouraged to use bash commands
  and
  python code to do that.

- You are encouraged to write down your thought and actions in `logs`
folder.
  This can help you keep track of your process and also provide useful
  information for future reference.
```