DBugScribe: Automatic Database Bug Reproduction from Community Reports

SUYANG ZHONG*, National University of Singapore, Singapore
MO SHA, Alibaba Cloud, Singapore
SHENG WANG, Alibaba Cloud, Singapore
FANGYUAN ZHOU, Alibaba Cloud, China
FEIFEI LI, Alibaba Cloud, China
KIAN-LEE TAN, National University of Singapore, Singapore

Major Database Management Systems (DBMSs) like MySQL field a high volume of bug reports daily. Yet, reproducing database bugs from natural language reports demands substantial developer effort due to manual interpretation and trial-and-error reconstruction. Database bugs are particularly challenging because they involve reconstructing multi-dimensional states that encompass configurations, schemas, data, queries, and validation oracles, while coping with natural language ambiguity. Existing automated bug reproduction techniques, designed for stateless functional testing, cannot address the stateful, multi-dimensional nature of database bugs. We present **DBugScribe**, the first framework specifically designed to automatically synthesize validated, executable, and structured reproduction scenarios for database bugs directly from users' natural language bug reports. At its core, **DBugScribe** introduces a domain-specific language (DSL) with formal semantics to represent bug scenarios as composable specifications. A novel hybrid synthesis approach integrates LLM-based information extraction with rule-based validation and self-refinement. Evaluated on 218 confirmed bug reports from eight recent DBMS testing tools covering MySQL, TiDB, and MariaDB, **DBugScribe** achieves 72.9% reproduction success. In our setup, **DBugScribe** can synthesize and execute a reproduction scenario within minutes per report. Beyond immediate reproduction, **DBugScribe** aggregates validated scenarios into a structured knowledge base that enables systematic bug analysis and cross-DBMS bug detection. This work transforms database bug reproduction from a manual, ad-hoc process into an automated and principled discipline, advancing both software engineering practice and database reliability research.

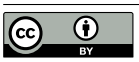
CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Software creation and management*; • **Information systems → Database management system engines**; • **Computing methodologies** → *Natural language processing*.

Additional Key Words and Phrases: bug reproduction, database systems, DBMS testing, natural language processing, domain-specific language, large language models, automated testing

ACM Reference Format:
Suyang Zhong, Mo Sha, Sheng Wang, Fangyuan Zhou, Feifei Li, and Kian-Lee Tan. 2026. **DBugScribe**: Automatic Database Bug Reproduction from Community Reports. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 157 (June 2026), 26 pages. https://doi.org/10.1145/3802034

*Suyang contributed to this work while interning at Alibaba Cloud.

Authors' Contact Information: Suyang Zhong, suyang@u.nus.edu, National University of Singapore, Singapore; Mo Sha, shamo.sm@alibaba-inc.com, Alibaba Cloud, Singapore; Sheng Wang, sh.wang@alibaba-inc.com, Alibaba Cloud, Singapore; Fangyuan Zhou, fory@alibaba-inc.com, Alibaba Cloud, China; Feifei Li, lifeifei@alibaba-inc.com, Alibaba Cloud, China; Kian-Lee Tan, dcstankl@nus.edu.sg, National University of Singapore, Singapore.



ACM 2836-6573/2026/6-ART157
https://doi.org/10.1145/3802034

1 Introduction

Database management systems (DBMSs) are among the most complex and critical software systems in operation today, with codebases often spanning millions of lines. This inherent complexity makes them prone to various categories of defects, including logic bugs [3, 41–43, 48, 49, 63] and crash bugs [20, 46, 66]. Despite decades of research and engineering effort, mainstream DBMSs continue to receive large volumes of bug reports every year. However, the process of handling these reports remains highly inefficient. Empirical studies show that developers spend nearly half of their overall programming time on debugging, with a non-trivial fraction of this effort dedicated specifically to reproducing and validating reported issues [1, 38]. Analyses of large-scale software repositories further indicate that a significant portion of reported bugs cannot be reliably reproduced [23, 39, 72], either because the bug exhibits nondeterministic behavior, the report omits key conditions required for reproduction, or the observed anomaly is in fact intended behavior. These challenges not only consume substantial engineering effort, but also delay the resolution of genuine defects in database systems that underpin countless data-driven applications.

Bug reports are valuable empirical evidence of how failures occur in database systems. They record concrete fault instances that can inform debugging, testing, and system hardening. However, a fundamental challenge remains: there exists a substantial gap between the way database bugs are reported and the way they need to be reproduced. Reports are typically written in natural language, often incomplete or ambiguous, and seldom include executable reproduction steps. Although platforms such as JIRA or GitHub collect detailed information about issue descriptions, severity, and resolution, this information is largely unstructured and inconsistent, limiting automated analysis and hindering the extraction of actionable insights. As a result, developers must manually interpret each report, reconstruct the necessary database environment, and attempt reproduction through labor-intensive trial and error.

Given these challenges, one might look to automated bug reproduction techniques for relief. However, existing automated bug reproduction frameworks focus on general-purpose software with well-defined APIs and stateless execution contexts, *e.g.*, LIBRO [25], AEGIS [53], and SWE-agent [56]. These approaches cannot handle the distinctive challenges of database systems, where failures depend on intricate dependencies among schema design and configuration settings. While recent bug-detection tools analyze large corpora of DBMS bugs [10, 13, 54], they primarily focus on manual categorization and statistical characterization rather than automated reproduction. This gap highlights the need for a systematic framework to translate bug descriptions into structured and automated reproduction pipelines.

Fortunately, recent advances in natural language processing offer a promising path to bridge this gap. Large Language Models (LLMs) have demonstrated the ability to parse unstructured text and translate informal descriptions into formal representations, *e.g.*, through Text-to-SQL systems [14, 19, 29, 60]. We leverage this capability for a novel purpose: applying it to debugging contexts, translating bug behavior descriptions into candidate executable queries combined with oracle formalization and multi-hypothesis refinement. Specifically, we treat a bug's complete reproduction context (schema design, initial data state, operation sequences, configuration parameters, and validation predicates) as a formal bug scenario that can be systematically synthesized and validated.

We present **DBugScribe**, a DSL-based framework that automates the reproduction and analysis of database bugs from natural language reports. At its core, **DBugScribe** introduces a domain-specific language (DSL) with formal semantics to represent bug scenarios as standardized, executable specifications. Coupled with Text-to-SQL-enhanced extraction, this DSL-based formalization enables an end-to-end automated pipeline from natural language bug reports to reproducible test cases and validated outcomes. As illustrated in Fig. 1, **DBugScribe** translates natural language reports into

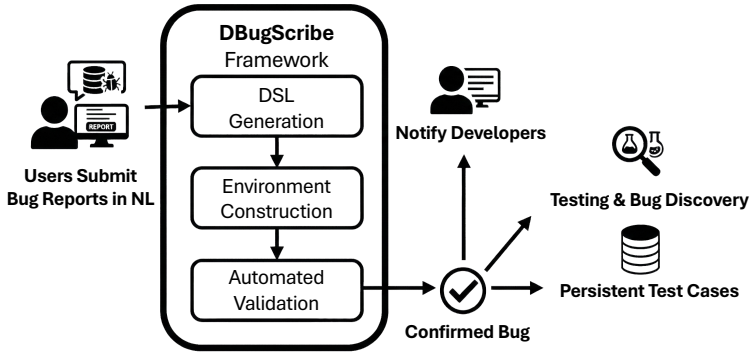


Fig. 1. Overview of **DBugScribe**.

formal **DBugScribe** DSL specifications, provisions containerized DBMS execution environments, performs automated reproduction and validation, and separates outcomes into two branches. The confirmed-bug branch triggers developer notifications, while the knowledge-base branch persists **DBugScribe** DSL scripts as regression test cases and formal bug scenario representations for systematic analysis. **DBugScribe** operates through an end-to-end pipeline that translates natural language reports into executable **DBugScribe** DSL scenarios, synthesizes and validates them via LLM-guided extraction and refinement, and executes them in isolated environments with oracle evaluation. The full architecture and pipeline details are presented in Section 3.

We evaluate **DBugScribe** on 218 real-world bug reports discovered by 8 state-of-the-art testing tools from recent research, covering MySQL, TiDB, and MariaDB. The framework achieves 72.9% reproduction success and can synthesize and execute a reproduction scenario within minutes per report in our setup, automating the transformation from natural-language reports to validated, executable scenarios. Beyond reproduction, **DBugScribe** supports systematic bug analysis through its unified DSL representation. It automatically synthesizes and validates self-contained reproduction scenarios, which can help standardize and streamline triage workflows when integrated into bug tracking. The persistent **DBugScribe** DSL scenarios enable systematic regression testing across versions, and successfully identified regression bugs in MySQL. Most notably, by retargeting synthesized scenarios across DBMSs, **DBugScribe** discovered 37 additional bugs (5 from MySQL, 10 from TiDB, 22 from MariaDB), including a new crash bug in MySQL that we reported to developers and received confirmation. These cross-DBMS testing capabilities reveal shared vulnerabilities across related systems and provide actionable insights: emerging DBMSs can proactively leverage known bugs from established systems, while researchers gain structured knowledge bases for systematic pattern analysis. Overall, **DBugScribe** advances both engineering practice and research understanding of DBMS reliability.

The contributions of this paper are summarized as follows:

- We establish the first formal model for database bug scenarios, treating them as first-class objects with rigorous execution semantics, dependency resolution, and compositional oracle predicates. We instantiate this formalization through the **DBugScribe** DSL that enables systematic representation and automated reasoning about DBMS-specific stateful behaviors.
- We develop **DBugScribe**, an end-to-end automated framework that bridges natural language bug reports to validated reproduction scenarios through LLM-based semantic extraction, Text-to-SQL enhanced oracle synthesis with iterative refinement, **DBugScribe** DSL synthesis with rule-based validation, and cross-DBMS execution with knowledge base accumulation.

- We evaluate **DBugScribe** on hundreds of real-world bug reports from recent state-of-the-art testing tools across multiple mainstream DBMSs, demonstrating promising reproduction success, minutes-level automated turnaround per report in our setup, and the ability to uncover recurring reliability patterns that advance DBMS testing research.

2 Preliminaries

2.1 Problem Formulation

We define automated database bug reproduction as follows.

Input: A natural-language bug report r describing anomalous behavior in a DBMS, potentially including free-form descriptions, SQL snippets, partial schema information, expected and observed outcomes, error messages, and version details.

Output: A self-contained, executable bug scenario d, specified in the **DBugScribe** DSL, that faithfully reproduces the reported behavior when executed under the target DBMS configuration.

General software bug reproduction typically involves test cases with a sequence of function calls and specific arguments. In contrast, database bug reproduction demands capturing multi-dimensional state and validation criteria. We identify four key differences:

(1) State dependence: Stateless function bugs can be reproduced via simple assertions such as `assert(f(x) == y)`. In contrast, database bugs require reconstructing complete system state. The bug manifests only when multiple conditions are simultaneously satisfied: specific query structures involving subqueries and joins, precise data values, particular constant representations in expressions, and the target DBMS version; omitting any element prevents reproduction. Such bugs depend on the interplay among schema design, data state, and workload context. This multi-dimensional dependency makes database bug reproduction significantly more challenging.

(2) Oracle complexity: General software often validates via simple return value checks or crash detection. Database bugs require more sophisticated oracle specifications. Logic bugs need to verify result correctness under relational semantics, often requiring explicit expectations or metamorphic relations (*e.g.*, semantically equivalent queries differing only in constant representation should yield identical results). Crash bugs require detecting abnormal termination and error conditions. This diversity necessitates flexible validation mechanisms beyond simple assertions.

(3) Environment sensitivity: While function bugs typically reproduce across environments, database bugs are highly sensitive to environmental variations. Variations in DBMS versions, collation settings, time zones, optimizer configurations, or resource limits can alter bug manifestation. For instance, different optimizer plans may cause bugs to appear or disappear, while encoding settings can affect string comparison results. Reproduction therefore requires precise environmental specification and isolation.

(4) Cross-system portability: General software test cases rarely transfer across systems because of incompatible APIs and substantially different software architectures. Database bugs, by contrast, often stem from shared SQL semantics despite dialect-level differences, making cross-DBMS reproduction possible. Moreover, many DBMSs share similar underlying architectures and optimization strategies (*e.g.*, MySQL and MariaDB), allowing related bugs to manifest across systems. Prior work [67] has demonstrated the potential for cross-DBMS test reuse. Nevertheless, automatically adapting real-world bug scenarios to different DBMSs remains an open challenge.

These four characteristics make database bug reproduction a distinct problem requiring specialized treatment beyond general-purpose techniques. Our framework addresses these challenges through formal scenario specification, automated synthesis, containerized execution, and cross-DBMS abstraction. Section 3 presents the complete architecture and end-to-end pipeline.

2.2 Scope and Assumptions

Our framework targets SQL bugs in relational DBMSs covering single-instance deployments, with a primary focus on logic bugs and crash bugs. We assume bug reports provide minimal essential information, including at least the offending query (or enabling description), observed unexpected behavior, and DBMS version. Our LLM-based synthesis handles typical incompleteness (*e.g.*, DBMS setup scripts or explicit execution sequence) through iterative refinement but fails when critical information is entirely absent. Our framework also aims to target non-deterministic bugs in a best-effort manner. Our framework explicitly excludes: (1) specialized hardware dependencies; (2) multi-node distributed deployments; (3) external system dependencies beyond standard installations; (4) pure concurrency race conditions where manifestation depends on thread interleaving. These constraints cover the majority of open-source bug reports while reflecting principled trade-offs between generality and automation reliability. We further discuss potential extensions beyond these current constraints in Section 5.

3 The DBugScribe Framework

3.1 Design Philosophy

Our framework combines neural semantic understanding with constraint validation for automated database bug reproduction. Three key design rationales guide this approach.

Rationale 1: DSL as Structured Intermediate Representation. Database bug scenarios exhibit stateful, multi-dimensional characteristics requiring coordinated specifications of schema evolution, data distributions, setup, and different validation methods. We introduce the **DBugScribe** DSL as a domain-specific structured intermediate representation tailored for these requirements, designed to be simultaneously expressive (captures complex database states) and declarative (specifies what to reproduce, not how). This design choice separates the concern of "what to reproduce" from "how to execute", facilitating automated reasoning about bugs through formal semantics and enabling potential cross-DBMS testing.

Rationale 2: Hybrid Synthesis Architecture. While LLMs excel at semantic understanding of natural language, they produce structurally invalid outputs in constrained domains: synthesizing bug scenarios from scratch yields type mismatches between schema and data, and referential integrity violations. Our hybrid architecture addresses this limitation by decomposing synthesis into complementary stages: neural extraction leverages LLM semantic understanding to parse unstructured reports into structured components, while rule-based validation enforces domain constraints through schema consistency checks, dependency resolution, and type inference. This separation of concerns ensures that generated scenarios are both semantically faithful to reported behaviors and formally valid with respect to database invariants—a property unattainable through pure neural generation or pure symbolic reasoning alone.

Rationale 3: Execution-Guided Self-Refinement. Bug scenario synthesis faces an inherent validity-correctness tension: initial synthesis may produce statically valid scenarios that fail dynamically (execution errors prevent completion) or produce false negatives (scenarios execute successfully but fail to trigger the reported bug, indicating semantic misalignment). Static validation alone cannot resolve this tension, as it verifies structural constraints without semantic bug-triggering properties. We address this through execution-guided self-refinement that closes the loop between synthesis and validation: execution feedback provides diagnostic signals (*e.g.*, error messages or oracle mismatches) that guide iterative scenario correction via LLM-as-a-judge [17, 65], progressively refining scenarios until they achieve both execution success and oracle satisfaction. This feedback-driven approach transforms synthesis from a one-shot generation problem into an iterative search process that converges toward minimal, triggering scenarios.

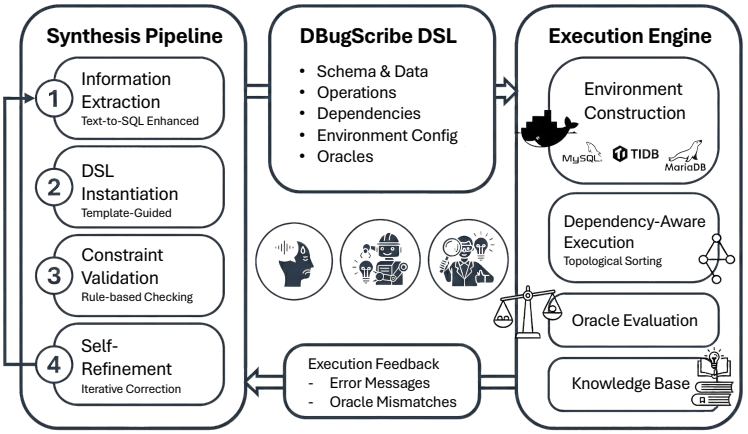


Fig. 2. **DBugScribe** system architecture.

3.2 System Architecture

Fig. 2 illustrates the overall architecture of **DBugScribe**, which comprises three core components organized around a central intermediate representation. At the heart of our framework lies the **DBugScribe** DSL (see Section 3.3), a structured intermediate representation that realizes Rationale 1's separation of "what to reproduce" from "how to execute." The **DBugScribe** DSL formalizes bug scenarios, capturing schema, data, operations, environment, and oracles in a declarative YAML-based syntax. The *synthesis pipeline* (see Section 3.4) transforms natural language bug reports into valid **DBugScribe** DSL scenarios through four stages: Text-to-SQL enhanced information extraction, DSL instantiation, rule-based validation, and execution-guided refinement. This realizes Rationale 2's hybrid architecture, combining neural semantic understanding of natural language with symbolic correctness guarantees. The *execution engine* (Section 3.5) consumes synthesized **DBugScribe** DSL scenarios by constructing isolated execution environments, executing operations with dependency-aware scheduling, and evaluating oracles. The **DBugScribe** DSL abstraction further enables cross-DBMS bug reproduction for differential testing. A knowledge base stores validated scenarios to support iterative improvement, facilitating bug pattern analysis and providing seeds for fuzzing.

The architecture establishes a complete data flow: natural language bug reports flow through the synthesis pipeline, producing **DBugScribe** DSL scenarios that execute in isolated environments; execution feedback guides iterative refinement when initial attempts fail; confirmed scenarios persist in the knowledge base, feeding back to improve future synthesis through few-shot learning and template reuse.

3.3 DBugScribe DSL: DBMS Bug Scenarios

We now detail the **DBugScribe** DSL that serves as the structured intermediate representation (Rationale 1), establishing formal foundations for bug scenario specification.

3.3.1 Motivation and Design Goals. Existing representations for database testing occupy two extremes. SQL test suites (*e.g.*, MySQL regression tests) provide concrete cases but lack structured metadata: schema dependencies are implicit, oracles are embedded in expected output files, and cross-DBMS portability requires manual adaptation. SQuaLity [67] represents an initial attempt to address the test reuse problem; however, its ability is limited, as it is hard to automatically

expand the test cases. Automated testing tools (*e.g.*, SQLsmith [46] and SQLancer [41–43]) provide automation test generation in bug discovery, but the generated scenarios are transient. Once a bug is detected, it is often difficult to deterministically reproduce the generation paths.

Our **DBugScribe** DSL navigates this trade-off through *declarative specifications with formal semantics*. Three design goals guide our approach, with measurable success criteria: **(1) Expressiveness**: our **DBugScribe** DSL covers schema definitions, data state, operation sequences, configuration parameters, and oracle predicates, thereby capturing all five components of the formal bug scenario representation. **(2) Simplicity**: a compact YAML-based syntax maintains human readability while enabling automated parsing, with scenarios averaging 100-120 lines for typical bugs. **(3) Abstraction**: scenarios focus on logical bug characteristics rather than DBMS-specific implementation details, which, as a byproduct permits attempting reproduction on alternative systems for differential analysis. The key insight guiding our design is that database bugs can be fundamentally characterized as *state transformations under constraints*—comprising an initial state (schema and data), an operation sequence (workload), environmental settings (configuration), and a validation predicate (oracle).

3.3.2 Formal Semantics. We formalize a bug scenario as a five-tuple $d = \langle \Sigma, D, W, \phi, \psi \rangle$ comprising:

Σ: **Schema** – A relational schema $\Sigma = (T, I, C)$ comprising tables, indexes, and integrity constraints (*e.g.*, primary keys, foreign keys, and uniqueness, as well as other recoverable integrity constraints when they can be extracted from the report).

D: **Data** – The initial database state, specified alongside Σ in `schema_data_steps`. $D(t)$ denotes the finite contents of table identifier t (*i.e.*, the set/multiset of tuples currently stored in t), and D maps table identifiers to their current contents; we use 2^{Tuple} as a concise notation for a finite collection of tuples in the scenario.

W: **Workload** – A sequence of SQL operations annotated with explicit dependencies (`depends_on`) that determine a valid execution order. Formally, $W = \langle (w_1, \delta_1), \ldots, (w_k, \delta_k) \rangle$, where each w_i is a SQL statement and $\delta_i \subseteq \{1, \ldots, i-1\}$ encodes acyclic dependencies.

ϕ: **Environment** – The execution context of the scenario, specified in `setup`, including the target DBMS version and configuration parameters that may affect bug manifestation. These parameters may include engine options, SQL modes, collation or locale settings, and other runtime settings needed to faithfully recreate the reported behavior.

ψ: **Oracle** – Validation predicates over execution outcomes, captured by `assertions`. Intuitively, ψ checks whether the observed behavior matches the expected symptom described in the report. Formally, we model $\psi : \text{Outcome} \to \{\top, \bot\}$, where an outcome $\text{Outcome} = (R, E, S)$ summarizes the query result R, error/exception message E, and execution status S.

Execution semantics. Conceptually, execution first installs the schema and data (Σ, D), then runs the workload W respecting dependencies, and finally evaluates the oracle ψ over the collected outcomes. Formally, $[\![d]\!]_e = \text{let } e' = \text{Install}(e, \Sigma, D) \text{ in } \text{Exec}(e', W)$, where Install applies DDL in Σ and loads D, and Exec executes W respecting dependencies.

Oracle types. We support two high-level oracle categories: (i) logic-bug oracles that check result correctness (including metamorphic relations over multiple operations), and (ii) crash-bug oracles that detect abnormal exceptions and error signatures. Precise predicate forms are omitted from the main exposition for readability and can be derived from the **DBugScribe** DSL assertion specification.

3.3.3 ***DBugScribe*** *DSL Specification.* We illustrate the **DBugScribe** DSL using a verified MySQL bug report from the MySQL bug forum[1] (#112394), where semantically equivalent queries involving subqueries and joins produce inconsistent results. Fig. 3 presents the complete synthesized **DBugScribe** DSL scenario for this report and serves as our reference throughout this specification.

A **DBugScribe** DSL scenario is a YAML document with five main sections: metadata, plus four sections realizing the formal five-tuple components (schema Σ and data D unified in `schema_data_steps`, workload W in `operations`, environment ϕ in `setup`, and oracle ψ in `assertions`). We now describe each section's structure and semantics, illustrating with the MySQL optimizer bug example.

Metadata includes a unique `bug_id` for identification and an optional `bug_pattern` label (*e.g.*, `query_variation`, `crash`) that guides template selection and facilitates systematic pattern mining across the bug corpus.

Setup (ϕ-realization) specifies environment configuration $\phi = (\text{dbms}, \text{version}, \text{config})$. The configuration includes: `dbms_type` (target DBMS), `dbms_version` (version), `docker_image` (bit-level reproducibility), and `connection_params` (database name, credentials, settings). The fields `docker_image` and `connection_params` collectively realize the config component of ϕ. This multi-level pinning addresses environment sensitivity (the primary cause of irreproducibility) while enabling regression testing by varying versions and differential testing by varying DBMSs.

Schema_data_steps (Σ- and D-realization) unify schema $\Sigma = (T, I, C)$ and data $D : T \to 2^{\text{Tuple}}$ into a dependency-ordered sequence. Each step specifies `step_id`, `action` (DDL: `create_table`, `create_index`; DML: `insert_data`), `sql_query`, and `depends_on` encoding the partial order. The unified representation enables fine-grained control over schema structure (e.g., testing with/without indexes), precise data distribution specification (NULL values, boundary cases), and scenario reuse through schema-data separation.

Operations (W-realization) correspond to the workload sequence $W = \langle (w_1, \delta_1), \ldots, (w_k, \delta_k) \rangle$. Each one specifies `step_id`, `sql_query` (w_i), `capture_result` flag (enables oracle evaluation; unnecessary for DML unless verifying side-effects), and `depends_on`. Formally, $\delta_i \subseteq \{1, \ldots, i-1\}$ captures inter-operation dependencies within W, where $j \in \delta_i$ indicates operation w_i depends on w_j's effects. At the implementation level, `depends_on` may additionally reference `schema_data_steps` identifiers to enforce execution ordering, though dependencies on Σ and D are semantically implicit through the execution model where $\text{Install}(\Sigma, D)$ precedes $\text{Exec}(W)$. This dependency mechanism accommodates diverse bug patterns and enables future extensions, including concurrency issues via independent operations ($\delta_i = \emptyset$), transactional bugs through explicit ordering ($j \in \delta_i$ for read-after-write dependencies), and isolation violations via deliberate dependency constraints.

Assertions (ψ-realization) encode oracle predicates through the mandatory `assertion_type` field, with values drawn from {`row_count`, `result_set`, `comparison`, `error`, `crash`}, together with `target_operation` and additional type-specific fields. The ψ_{exact} oracle requires an `expected_value` field for the expected result. The ψ_{meta} oracle requires `operation_ids` and a `comparison_operator` field for relations such as `equals` and `not_equals`. These assertion types cover cardinality expectations (`row_count`), exact result checking under bag/sequence semantics (`result_set`), query-variation bugs via metamorphic relations (`comparison`, where `equals` expects identical results), error status and message matching (`error`), and server unavailability detection (`crash`). Conjunctive composition, $\psi = \bigwedge_i \psi_i$, enables multi-faceted correctness checking.

This example demonstrates the five-tuple formalization instantiated on a concrete bug scenario. Schema $\Sigma = (T, I, C)$ comprises table $T = \{\texttt{t0(vkey int)}\}$ with $I = C = \emptyset$, achieving minimal structure. Data instance $D(\texttt{t0}) = \{(5)\}$ contains one tuple. Workload W contains two

[1]"Inconsistent results caused by subqueries in FROM and EXISTS". Reported by Zuming Jiang, Sep 2023. Status: Verified. https://bugs.mysql.com/bug.php?id=112394

```yaml
# Metadata
metadata:
  bug_id: mysql-112394
  bug_pattern: query_variation
# Setup (phi): Environment
setup:
  dbms_type: mysql
  dbms_version: 8.0.34
  docker_image: mysql:8.0.34
# Schema (Sigma) & Data (D)
schema_data_steps:
- step_id: schema_001
  action: create_table
  sql_query: create table t0 (vkey int);
  depends_on: []
- step_id: data_001
  action: insert_data
  sql_query: insert into t0 (vkey) values (5);
  depends_on: [schema_001]
# Operations (W)
operations:
- step_id: op_001
  sql_query: select 1 as c0 from ((select case when true then 0 else 0 end as c_0 from t0 as ref_5)
      as subq_2 right outer join t0 as ref_6 on (subq_2.c_0 = ref_6.vkey)) where exists (select 1
     from t0 as ref_9 where (subq_2.c_0 <> ref_9.vkey));
  depends_on: [data_001]
- step_id: op_002
  sql_query: select 1 as c0 from ((select 0 as c_0 from t0 as ref_5) as subq_2 right outer join t0
      as ref_6 on (subq_2.c_0 = ref_6.vkey)) where exists (select 1 from t0 as ref_9 where (subq_2.
     c_0 <> ref_9.vkey));
  depends_on: [data_001]
# Assertions (psi)
assertions:
- assertion_type: comparison
  target_operation: op_001
  operation_ids: [op_002]
  comparison_operator: equals
  description: Metamorphic oracle expecting semantically equivalent queries to yield identical
     results; bug detected when results differ
```

Fig. 3. **DBugScribe** DSL example for MySQL Bug #112394. Redundant information is omitted for clarity.

queries w_1, w_2 differing in constant representation, with $\delta_1 = \delta_2 = \emptyset$ (no inter-operation dependencies; YAML `depends_on` references to `data_001` are implementation-level execution ordering, as dependencies on Σ and D are implicit through $\text{Install}(\Sigma, D)$ preceding $\text{Exec}(W)$). Environment $\phi = (\text{mysql}, 8.0.34, \text{config})$ pins DBMS version and configuration. Oracle $\psi_{\text{meta}}(R_1, R_2, \mathcal{T})$ verifies multiset equality $R_1 \equiv_{\text{bag}} R_2$ for results from queries related by transformation $\mathcal{T}$ (constant folding). The bug manifests when $\psi_{\text{meta}} = \bot$ due to incorrect optimizer handling: w_1 yields $R_1 = \{(1)\}$ while w_2 yields $R_2 = \emptyset$, violating the expected $\psi_{\text{meta}} = \top$ under semantic equivalence.

3.4 LLM-Driven Synthesis Pipeline

The synthesis pipeline transforms a natural language bug report into a valid **DBugScribe** DSL scenario (as defined in Section 3.3) that faithfully reproduces the reported behavior. This transformation must bridge the semantic gap between informal descriptions and formal specifications. For a

report r, we seek to synthesize a scenario d that satisfies structural validity constraints (well-formed schema, type-consistent data, and acyclic operation dependencies) while successfully triggering the bug through execution.

This synthesis task presents multiple challenges. Pure LLM generation is prone to validity violations, including invalid syntax of the DSL, missing steps, or wrong format of the assertion. Moreover, even structurally valid scenarios may fail to reproduce bugs due to incomplete information—reports often omit critical details, for example, the database setup scripts (*e.g.*, `CREATE DATABASE`).

To overcome these challenges, we propose a four-stage hybrid architecture (Rationale 2) that integrates LLM-based extraction with domain-specific validation and execution-guided feedback. Instead of generating full scenarios in a single step, our method incrementally builds them, adding only essential components (*e.g.*, tables referenced in queries) to ensure practical minimality.

Stage 1: Text-to-SQL Enhanced Information Extraction. LLM π parses report r into structured intermediate representation IR(r) comprising DBMS version, schema/data hints, queries, and symptoms. Unlike conventional extraction that produces loose textual hints, we integrate a Text-to-SQL module that directly generates candidate executable SQL statements from behavior descriptions (*e.g.*, "joining X and Y yields incorrect counts" → multiple JOIN query candidates with COUNT predicates). For the MySQL optimizer bug example (Fig. 3), the extracted IR includes: `dbms`: "MySQL v8.0.34", `schema_hints`: {table `t0` with `INT` column `vkey`}, `data_hints`: {single row with `vkey = 5`}. Beyond these structural elements, `candidate_queries` contains multiple ranked SQL statements representing query variations (*e.g.*, with `CASE` expression vs. constant, different join orders) that exhibit the described behavior.

Alongside queries, the module formalizes symptoms into oracle predicates: for example, a bug report stating "expected 1 row, got 3" would yield an exact oracle `ASSERT row_count = 1`, while the MySQL optimizer bug's symptom "semantically equivalent queries produce different results" yields a metamorphic comparison oracle (as shown in Fig. 3). Candidates are ranked by structural matching (table/column mentions), oracle plausibility (row count issues favor COUNT queries), and template affinity. Top candidates proceed to Stage 2; lower-ranked ones are retained as fallbacks for Stage 4 if primary candidates fail to trigger bugs. This multi-hypothesis strategy leverages natural language ambiguity for validity and introduces a novel application of NL2SQL in debugging contexts, distinct from its conventional use in question answering.

Stage 2: DSL Instantiation. Instead of generating **DBugScribe** DSL scenarios from scratch, we leverage constraint-guided instantiation of parameterized templates encoding common bug patterns. **DBugScribe** instantiates a bug scenario as a five-tuple $d = \langle \Sigma, D, W, \phi, \psi \rangle$ (Section 3.3). In our implementation, Stage 2 invokes the LLM with constrained decoding to instantiate DSL templates from Stage 1's intermediate representation. The LLM outputs a structured specification (*e.g.*, JSON) for schema, data, operations, and oracle fields, which we then deterministically map to the corresponding **DBugScribe** DSL template.

Stage 3: DSL Validation. A validator completes partially-specified scenarios through three mechanisms: **(1) Schema inference** from type requirements (*e.g.*, inferring `t0.vkey` is `INT` from usage in comparisons and joins); **(2) SQL sanity checking** through a best effort parser; **(3) Dependency ordering** via topological sort on the dependency graph constructed from $\{\delta_1, \ldots, \delta_k\}$. Stage 3 performs this static checking deterministically and reports violations; any necessary repairs are handled in Stage 4 using execution feedback and LLM-guided refinement.

Stage 4: Self Refinement. We employed execution feedback and LLM self-reflection to iteratively refine scenarios for correctly reproducing the bugs. If an operation fails (*e.g.*, "`column X not found`") unexpectedly, we extract diagnostics and prompt the LLM to propose fixes (*e.g.*, "Add column X with appropriate type"). The LLM generates patches (updating IR for re-synthesis via

Stages 2-3) to produce refined d'. The refinement loop iterates up to a maximum limit, with most successful cases converging within a few iterations. **Execution errors** trigger this loop; **Oracle mismatches** (operations executes successfully but ψ evaluates to $\top$, indicating the bug does not reproduce) receive oracle-guided query refinement: we retry with lower-ranked candidate queries from Stage 1's list (up to N candidates, typically multiple alternatives), swapping queries while maintaining schema/data. The LLM reasons about why the bug fails to reproduce (*e.g.*, index not used, data insertion missing) and proposes adjustments of operations and schema or data steps. This tight feedback loop, where the same Text-to-SQL module participates in both initial extraction and iterative refinement, enables query and assertion-level repairs. We also prompt the LLM to identify whether the generated assertion aligns with the reported symptom; if misalignment is detected, we re-extract the oracle from the report and re-synthesize the scenario. The four-stage architecture exploits complementary strengths: Stage 1 extracts semantic information from natural language reports; Stage 2 instantiates parameterized templates encoding domain knowledge; Stage 3 validates structural correctness through schema inference and dependency resolution; Stage 4 refines scenarios via LLM self-reflections. Fig. 3 illustrates the complete transformation from an informal community bug report to an executable **DBugScribe** DSL scenario for the MySQL optimizer bug. From a performance perspective, LLM API calls dominate latency (Stages 1, 2, and 4), while the deterministic parts of Stages 2–3 (template matching and static validation) typically execute in milliseconds. Though further optimization is possible (*e.g.*, prompt caching, batching), current performance suffices for offline workflows where automated synthesis can produce executable reproduction artifacts at scale.

3.5 Execution and Validation Engine

The execution engine translates synthesized **DBugScribe** DSL scripts into concrete bug reproductions, ensuring reproducibility through environment isolation.

3.5.1 Dependency-Aware Execution Model. Given the workload sequence $W = \langle (w_1, \delta_1), \ldots, (w_k, \delta_k) \rangle$, we construct a dependency graph $G = (V, E)$ where $V = \{1, \ldots, k\}$ and edges represent precedence constraints: $(i, j) \in E$ iff $i \in \delta_j$, meaning w_i must execute before w_j. We verify that G is acyclic (as required by the formalization) during DSL synthesis; if a cycle is detected, generation is aborted with a diagnostic identifying the conflicting operations, which is reported as a synthesis error. For valid acyclic graphs, topological sort on G yields an execution order $\sigma : \{1, \ldots, k\} \to \{1, \ldots, k\}$ respecting all dependencies. The execution follows the formal semantics: operations execute sequentially in the order determined by σ, where $\text{Exec}(e', W)$ produces outcomes $\{o_{\sigma(1)}, \ldots, o_{\sigma(k)}\}$ and each $o_i = (R_i, E_i, S_i)$ results from executing w_i. If an operation fails, dependent operations are skipped; independent operations continue, enabling partial execution and detailed failure diagnostics.

3.5.2 Oracle Evaluation Strategies. For logic bugs, we evaluate ψ_{exact} or ψ_{meta} as defined in the formal semantics. The MySQL example evaluates the metamorphic oracle $\psi_{\text{meta}}(R_1, R_2, \mathcal{T}) = \mathbb{1}[R_1 \equiv_{\text{bag}} R_2]$ where R_1 and R_2 are results from semantically equivalent queries. Under correct DBMS behavior, $\psi_{\text{meta}} = \top$; when $\psi_{\text{meta}} = \bot$ (indicating $R_1 \not\equiv_{\text{bag}} R_2$), the bug manifests as a violation of query semantic equivalence. For crash bugs, we instead evaluate $\psi_{\text{crash}}(S, E)$ where status S = crash is detected via non-zero exit codes, and pattern matching on E identifies specific failures such as "lost connection" or "segmentation fault".

3.5.3 Cross-DBMS Execution. The target DBMS of each scenario is specified by `dbms_type` in ϕ. As detailed in Section 3.3, the **DBugScribe** DSL's declarative design enables experimental retargeting to alternative DBMSs for differential analysis. Scenarios can be adapted by modifying the `dbms_type` and `dbms_version` fields while preserving the original bug scenario. Because DBMSs differ in syntax

and semantics, retargeting may lead to execution failures, behavioral differences, or unsuccessful reproductions. Nevertheless, when a scenario triggers the same bug on an alternative DBMS as on the original one, the observed behavior provides stronger evidence that the issue reflects a genuine bug rather than an artifact of a single implementation. We conducted preliminary experiments and empirical analysis (Section 4.4.5) to evaluate the effectiveness of this approach.

3.5.4 Environment Construction. Each scenario runs in isolation to prevent cross-test interference. We maintain version-specific environments for each DBMS (MySQL, TiDB, MariaDB), preloaded with versions and configurations defined in ϕ. The execution proceeds as follows: (1) select the environment manager matching `dbms_type` and `dbms_version`; (2) launch the environment (*e.g.*, docker container) with health checks verifying DBMS readiness; (3) execute **DBugScribe** DSL steps sequentially following dependency order; (4) collect results, error logs, and exit codes; (5) terminate the environment to ensure a clean state. Environment health checks employ database-specific mechanisms (*e.g.*, `mysqladmin ping` for MySQL) to guarantee service availability before operation execution. Environment startup incurs overhead (see Section 4.3 for timing analysis), which is acceptable for offline reproduction workflows that prioritize correctness and reproducibility over speed. For large-scale evaluation, we parallelize across multiple environments to leverage available compute resources.

Summary. The Execution and Validation Engine provides three key capabilities. **Environment isolation** is achieved through containerization, ensuring reproducible execution free from cross-test interference and state carryover. **Faithful reproduction** prioritizes accurate bug reproduction on the target DBMS specified in bug reports, with experimental support for differential testing on alternative systems to identify cross-DBMS vulnerabilities. **Compositional oracles** are fulfilled by the oracle evaluation strategies: logic bugs use explicit result comparison or metamorphic testing, crash bugs monitor process termination; these are composable through conjunctive predicates ($\psi = \bigwedge_i \psi_i$) as formalized earlier. The dependency-aware execution model respects ordering constraints encoded in the **DBugScribe** DSL, ensuring operations execute in topologically valid sequences satisfying all precedence requirements. Together, these components form a robust infrastructure that translates validated **DBugScribe** DSL scripts into reliable bug reproductions, completing the end-to-end pipeline from natural language reports to confirmed bug reproduction.

4 Experimental Evaluation

We evaluate **DBugScribe** in terms of end-to-end reproduction success on real-world database bugs from natural-language reports, together with efficiency and an internal ablation analysis of key pipeline components. We implement **DBugScribe** in Python (~5.7K LOC). All experiments are conducted on a server with a 64-core AMD EPYC 7763 processor (2.45 GHz) and 512 GB RAM, running Ubuntu 22.04. We access LLMs via the GPT-5 API[2]. Unless otherwise specified, we use GPT-5 mini for Stage 1 (Parsing) and GPT-5 for Stages 2 and 4 (Instantiation and Refinement); Stage 3 is deterministic and does not invoke an LLM. Internet access is disabled during LLM calls to prevent data leakage. For DBMS execution environments, we use Dockerized instances for MySQL and MariaDB, and TiUP[3], TiDB's package manager, for TiDB. All target DBMS versions are pre-installed before the reproduction experiments.

4.1 Dataset Construction and Curation

To evaluate **DBugScribe**, we constructed *BugBench*, a curated dataset of bugs from three relational DBMSs: MySQL, TiDB, and MariaDB. Our methodology for constructing *BugBench* was guided by

[2]https://openai.com/gpt-5

[3]https://docs.pingcap.com/tidb/stable/tiup-overview/

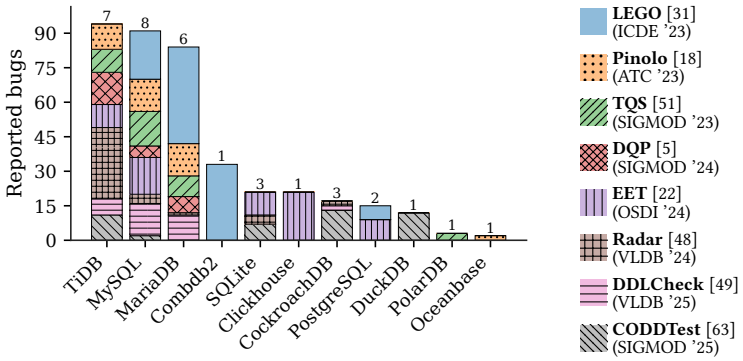


Fig. 4. Testing targets of recent DBMS testing tools.

Table 1. Bug Reproduction Dataset Statistics across DBMSs

DBMS	Collected	Reported	Conf.	Fixed	Dupl.	FP	N/A
MySQL	62	91	53	6	3	0	29
TiDB	91	94	53	34	4	3	0
MariaDB	65	84	20	12	33	2	17
Total	**218**	**269**	**127**	**51**	**40**	**5**	**46**

the principle of collecting bugs that are both recent and significant enough to be of interest to the database research community.

Selection Criteria. Rather than randomly sampling from public trackers, we source bugs from recent state-of-the-art database testing tools, which helps ensure non-triviality and prior vetting by researchers. We prioritize researcher-submitted reports because: (i) they are more likely to describe real DBMS bugs rather than usage questions or configuration issues; (ii) they often target core DBMS functionalities (rather than application-level concerns), aligning with the database community; and (iii) our selection spans three reporting platforms (e.g., GitHub, Jira, and MySQL forums) and eight DBMS testing tools, capturing diverse reporting formats for evaluating **DBugScribe**. We include tools that (1) were published at top-tier database (SIGMOD, VLDB, ICDE) or systems (OSDI, SOSP, ATC, EuroSys) venues in the past three years, (2) provide publicly accessible reports with explicit developer confirmation (excluding tools without public links or identifiers), and (3) target relational DBMSs (excluding NoSQL systems such as graph DBMSs).

Tool Survey. Applying the above criteria, we identified in total eight recent tools [5, 18, 22, 31, 48, 49, 51, 63] that reported confirmed bugs in widely used open-source relational DBMSs. Among surveyed papers, we excluded three tools [11, 36, 71] that aim for NoSQL systems (*e.g.*, Graph DBMSs), three tools [9, 10, 15] that aim for detecting transactional anomalies, and four tools [13, 32, 54, 55] without publicly available links or artifacts related to the bug reports. Fig. 4 shows each tool's targets and the number of reported bugs. MySQL, TiDB, and MariaDB are the most frequently targeted systems, each tested by at least six tools and averaging 86 reported bugs. As noted above, the resulting reports span multiple bug-tracking platforms and reporting styles, providing a diverse basis for evaluating **DBugScribe**. We therefore selected these three DBMSs for constructing *BugBench*.

Table 2. Bug Reproduction Performance across Methods

Method	Success Rate (%)				F→P (%)	Time (s)
	Overall	MySQL	TiDB	MariaDB		
Naïve	47.2	69.3	41.7	33.8	86.3	110.9
OneStage	49.0	74.1	42.8	33.8	96.2	160.5
W/o Refine	63.3	79.0	50.5	66.1	88.9	127.7
DBugScribe	72.9	88.7	61.5	73.8	100.0	274.3

Dataset Summary. Table 1 summarizes the composition of our curated *BugBench*. We inspected 269 reported bugs across the three DBMSs (**Collected**: 91 MySQL, 94 TiDB, 84 MariaDB). Of these, 218 had sufficient information for reproduction, including bugs confirmed (**Conf.**) or fixed by developers, as well as duplicates (**Dupl.**). Five reports were identified as false positives (**FP**) and excluded. Another 46 were unavailable (**N/A**) due to broken links provided or security restrictions (*e.g.*, MySQL hides details of crash bugs). Bugs reported to MariaDB exhibit a notably high duplicate rate, substantially increasing triage burden. The **DBugScribe** DSL representation provides a structured basis for duplicate analysis (*e.g.*, clustering bug scenarios by structural similarity of bug information), which may support future duplicate detection and triage.

4.2 Reproduction Effectiveness Analysis

We evaluate **DBugScribe** on 218 real-world DBMS bugs from *BugBench* with sufficient reproduction information, reporting end-to-end reproduction success and an internal ablation analysis of key pipeline components, as well as efficiency and failure patterns.

Methodology. We measure reproduction success rate as the percentage of bugs that **DBugScribe** successfully reproduces and validates against developer-confirmed outcomes. Since existing automated bug reproduction methods for general software (*e.g.*, AEGIS [53], LIBRO [25]) lack structured DSL support for DBMS-specific assertions and setup steps, we compare **DBugScribe** against three ablated variants that progressively disable key components of our four-stage synthesis pipeline. **Naïve** performs single-pass DSL generation without execution feedback or refinement, using only the initial extraction and instantiation stages. **OneStage** extends Naïve by incorporating self-refinement (Stage 4) but still relies on a single LLM pass for initial synthesis. **W/o Refine** includes all stages except execution-guided self-refinement, using only the initial synthesis attempt from Stages 1–3. We further report **DBugScribe**'s automated turnaround time as a system-level latency metric. We also summarize community confirmation latencies in our dataset as contextual evidence of end-to-end workflow latency; we do not interpret these latencies as human reproduction time, nor as time saved by **DBugScribe**.

Reproduction Success. Table 2 reports the end-to-end reproduction success rates of **DBugScribe** on *BugBench*, along with internal ablation baselines across three DBMSs. The **F→P** column reports the percentage of DSL scenarios that fail on the buggy version but pass on the latest version, thereby confirming correct bug reproduction. **DBugScribe** reproduces 72.9% of bugs overall, substantially outperforming OneStage (49.0%), W/o Refine (63.3%), and Naïve (47.2%). The one-stage LLM often fails to capture the multi-dimensional requirements of DBMS bugs (*e.g.*, schema, setup, and assertions) in a single pass. Refinement improves over both the one-stage baseline and the non-refinement variant and increases the quality of generated **DBugScribe** DSL scenarios. It achieves a 100% F→P rate; in contrast, W/o Refine attains only 88.9%, indicating that some reproduced assertions do not match the reported symptoms. We excluded one case from the F→P calculation

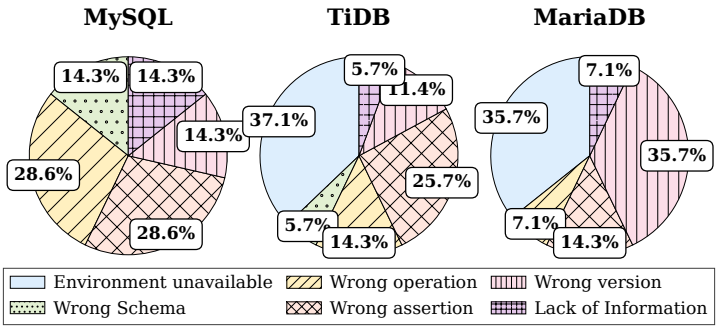


Fig. 5. Distribution of failure reasons across three DBMSs.

due to its classification as a regression bug (Section 4.4.4). By DBMS, **DBugScribe** achieves success rates of 88.7% on MySQL, 61.5% on TiDB, and 73.8% on MariaDB.

Efficiency. Community bug confirmation latencies for the reports in our dataset are often on the order of days to weeks. For example, only 21–29% of reports are confirmed within 24 hours (MySQL and TiDB), while 48% of MariaDB reports take more than a week. Although these latencies reflect a combination of factors in real-world workflows and are not directly attributable to the human effort required for reproduction, we view them as contextual evidence of end-to-end workflow delay. In contrast, we report **DBugScribe**'s automated turnaround time as pipeline latency: in our setup, **DBugScribe** synthesizes and executes a **DBugScribe** DSL scenario within 10 minutes on average (274.3 seconds; Table 2). If integrated into a bug-tracking workflow, **DBugScribe** can process newly filed reports in the background and attach a concrete, executable artifact to support downstream triage and discussion.

Failure analysis. We manually analyzed unsuccessful cases and grouped them into six categories (Fig. 5): environment unavailable, wrong operation, wrong assertion, wrong version, wrong schema, and lack of information. The most common failure reason is environment unavailability, which often stems from special build requirements—five MariaDB cases require AddressSanitizer—which **DBugScribe** does not yet support. Wrong operation, or schema failures arise when SQL is imperfectly mapped to **DBugScribe** DSL steps. Wrong assertion failures occur when the generated assertions do not accurately capture the expected behavior described in the bug report. For example, a TiDB bug [4] describes an incorrect error message when executing a query; however, **DBugScribe** failed to generate the pattern that matches the expected error message, leading to a failed reproduction. Lack of information cases generally come from reports with incomplete SQL or context, since users reported cases through attachments or external links that **DBugScribe** cannot access.

Human feedback. We discuss the potential benefits of incorporating minimal human feedback into the refinement loop, informed by our failure analysis (see Fig. 5). Below, we categorize the 56 unreproduced bugs by their root causes, ordered by the level of human effort likely required (from minimal to substantial): (1) **Wrong version** (10 cases): a small correction of version information would enable successful reproduction; (2) **Wrong schema** and (3) **Wrong operation** (11 cases in total): brief clarification of missing or misinterpreted schema definitions or operations could refine the scenario; (4) **Wrong assertion** (13 cases): human guidance could refine the oracle by clarifying expected behavior; (5) **Environment unavailable** (18 cases): reproduction requires substantial

[4]https://github.com/pingcap/tidb/issues/46538

(a) User report

```
### Affected version
- 5.5.61
- 10.11.1
...
```

(b) DBugScribe DSL

```
# ...
setup:
-   - docker_image: mariadb:10.11.10
+   - docker_image: mariadb:10.11.1-jc
```

Fig. 6. Repairing a failure with human interaction.

Table 3. Bug Reproduction Success by Bug Types

DBMS	Logic			Error			Crash		
	Succ.	Total	%	Succ.	Total	%	Succ.	Total	%
MySQL	52	59	88.1	3	3	100.0	–	–	–
TiDB	34	58	58.6	11	17	64.7	11	16	68.8
MariaDB	19	27	70.4	4	5	80.0	25	33	75.8

manual environment setup (e.g., building from source), which goes beyond light feedback; (6) **Lack of information** (4 cases): missing critical details make reproduction infeasible without timely, report-specific follow-up clarification from the original reporter.

Fig. 6 presents a case where **DBugScribe** initially failed to reproduce a MariaDB bug due to misinterpreting the version information in the report. The original report (Fig. 6(a)) lists two affected versions: 5.5.61 and 10.11.1. When generating setup for the **DBugScribe** DSL, **DBugScribe** incorrectly parsed the version 10.11.1 as 10.11.10 (Fig. 6(b), red block), leading to a failed reproduction. This is because the docker image provided by MariaDB for version 10.11.1 is tagged as `10.11.1-jc`, which **DBugScribe** failed to infer from the report. With minimal human intervention—simply correcting the docker image tag in the **DBugScribe** DSL (Fig. 6(b), green block)—**DBugScribe** successfully reproduced the bug. Incorporating such human feedback into **DBugScribe**'s iterative refinement process could further enhance its robustness and success rate, especially for environment-related issues.

4.3 Fine-Grained Performance Analysis

We further analyze **DBugScribe** effectiveness and efficiency through two perspectives, success by bug type and runtime breakdown across pipeline components. First, we examine reproduction results across different bug types to understand the expressiveness of the **DBugScribe** DSL. We can categorize bugs into three types: logic bugs (*e.g.*, incorrect query results), error-handling bugs (*e.g.*, unexpected error messages), and crash bugs (*e.g.*, server crashes). These types require different assertions and operations in the **DBugScribe** DSL, testing its versatility. Here, we regard error-handling bugs as a special kind of logic bugs as defined in Section 3.3.2 which return a wrong exception instead of a wrong result. Second, we analyze the runtime characteristics of **DBugScribe** to identify potential bottlenecks and areas for optimization. We break down the total time taken to process each bug report into three main components: parsing the natural language report, generating the **DBugScribe** DSL scenario (including refinement), and executing the scenario against the target DBMS.

Success by Bug Type. Table 3 presents success rates across logic, error-handling, and crash bugs. **DBugScribe** achieves its highest success on logic bugs (72.9% overall), with strong performance on MySQL (88.1%), TiDB (58.6%), and MariaDB (70.4%). Error-handling bugs are more challenging (72.0% overall) because they require **DBugScribe** to precisely capture exception conditions and validate error messages. For MySQL, details of crash bugs are hidden for security, so our dataset

Table 4. Average Time Consumption for Each Pipeline Stage per DBMS (Seconds and Percentage)

DBMS	Parsing		Generation		Execution		Total (Sec.)
	Sec.	%	Sec.	%	Sec.	%	
MySQL	31.9	14.2	180.0	80.3	12.2	5.5	224.2
TiDB	26.2	8.2	199.8	62.3	94.7	29.5	320.7
MariaDB	38.9	14.4	214.5	79.4	16.8	6.2	270.4

Table 5. Average LLM usage (tokens) and estimated cost (USD) per DBMS (per report)

	Parsing			Instantiation			Refinement			Total
DBMS	In.	Out.	$	In.	Out.	$	In.	Out.	$	$
MySQL	1,859	1,843	0.004	5,497	5,496	0.062	5,917	4,978	0.057	0.123
TiDB	1,935	1,619	0.004	5,783	6,021	0.067	5,574	5,141	0.058	0.130
MariaDB	3,346	2,051	0.005	6,487	7,427	0.082	6,678	5,995	0.068	0.156

contains no crash cases for MySQL. Crash bugs are generally straightforward to detect; across TiDB and MariaDB, we achieve 73.5% overall, though some cases still require specific environment setups (*e.g.*, ASan builds) that **DBugScribe** does not yet support. Overall, these results demonstrate the **DBugScribe** DSL's expressiveness in capturing diverse bug characteristics and the challenges in accurately modeling complex error conditions.

Runtime Breakdown. We report the average time to reproduce a bug and the time spent in each pipeline component (parsing, **DBugScribe** DSL generation with refinement, and execution); see Table 4. Across DBMSs, total time per bug averages 224.2s (MySQL), 320.7s (TiDB), and 270.4s (MariaDB). **DBugScribe** DSL generation dominates runtime, consuming 62–80% of total time, while parsing and execution account for 8–14% and 6–30%, respectively. TiDB exhibits substantially higher execution overhead (94.7s, 29.5%) due to its distributed architecture setup. In summary, the results indicate that while **DBugScribe** is efficient in automating bug reproduction, there is room for optimization, particularly in the **DBugScribe** DSL generation phase (*e.g.*, parallelizing LLM calls for each component).

Monetary Analysis. Table 5 reports the average LLM usage and estimated monetary cost per bug report on *BugBench*, broken down by DBMS and by pipeline stages that invoke the LLM: **Stage 1** (Parsing), **Stage 2** (Instantiation), and **Stage 4** (Refinement). **Stage 3** is deterministic (Validation) and thus incurs no LLM cost. We compute the dollar cost by aggregating the recorded input/output token counts for each stage and converting them to USD using the GPT-5 API pricing at the time of the experiments. In our setup, Stage 1 uses a cheaper model (GPT-5 mini), while Stages 2 and 4 use the full GPT-5 model, which explains the lower parsing cost relative to instantiation and refinement.

4.4 Extended Applications and Discoveries

By capturing bugs as persistent, executable **DBugScribe** DSL scenarios rather than ephemeral manual steps, **DBugScribe** enables systematic quality assurance capabilities and cross-system bug discovery previously infeasible with manual reproduction.

4.4.1 Flaky Bug Reproduction. To demonstrate **DBugScribe**'s ability to handle complex and even non-deterministic bugs, we present a case study of a flaky bug from TiDB[5] (see Fig. 7(a)). This bug was found by DQP [5] and marked as critical by developers. The original bug report provides

[5]https://github.com/pingcap/tidb/issues/46580

(a) Ambiguous user report

```
### Minimal reproduce step
CREATE TABLE t0(c0 INT);
...
SELECT /*+ MERGE_JOIN(t1, t0, v0)*/v0.c2, t1.c0 FROM v0,  t0...

### What did you see instead
Executing the above test case multiple times and the results are not the same.
Sometimes it returns 4 rows. Sometimes it returns empty: {}.
```

(b) DBugScribe DSL

```
# ...
operations:
  - step_id: op_run1
    sql_query: SELECT /*+ MERGE_JOIN(t1, t0, v0)*/ ...;
  - step_id: op_run2
    sql_query: SELECT /*+ MERGE_JOIN(t1, t0, v0)*/ ...;
  - step_id: op_run3
    sql_query: SELECT /*+ MERGE_JOIN(t1, t0, v0)*/ ...;
  - ...
assertions:
  - step_id: assert_flaky
    target_operation: op_run1
    type: comparison
    comparison_operator: not_equals
    operation_ids: [op_run1, op_run2, op_run3, ...]
```

Fig. 7. Synthesizing a **DBugScribe** DSL scenario for a flaky TiDB bug.

reproduction SQL statements and a brief, ambiguous description of the failure: *Executing the test case multiple times and the results are not the same.* This type of report is difficult to diagnose, as it requires repeated executions to observe the failure. As shown in Fig. 7(b), **DBugScribe** successfully interprets the ambiguous report and synthesizes a structured **DBugScribe** DSL scenario that captures the essence of the flaky behavior. This generated **DBugScribe** DSL includes multiple operations to repeatedly execute the same SQL query for mimicking the non-deterministic behavior. The assertion section specifies a `comparison` assertion that checks for differing results across these operations to capture the flakiness. We emphasize that flaky tests have been a long-standing challenge in general software research [27, 40] and **DBugScribe** uses a best-effort strategy based on bounded repetitions and cross-run comparisons: it targets flaky behaviors that manifest with non-trivial probability within a practical repetition budget, and it is not intended to cover extremely rare flakes (*e.g.*, once in 1K or 1M runs).

DBugScribe can reproduce certain flaky bugs from ambiguous descriptions by synthesizing bounded-repetition scenarios.

4.4.2 Automatic Bug Analysis. **DBugScribe** enables automated bug analysis through its structured, executable DSL representation. Recent work [13, 54] highlights the benefits of mining historical bug patterns to inform automated testing tools. **DBugScribe** facilitates this direction by providing standardized, machine-readable bug scenarios that can be systematically analyzed. For example,

Table 6. SQL Function Usage by Bug Type

Type	Bugs	# Funcs	Top Functions (# occurrence)
Logic	105	39	`cast` (10), `date_add` (8), `date_sub` (4)
Error	18	6	`cast` (5), `reverse` (2), `acos` (1)
Crash	36	29	`strcmp` (2), `hex` (2), `from_days` (2)

given bugs successfully reproduced in the **DBugScribe** DSL, one can prompt an LLM: "*Implement a script that identifies and counts the usage of SQL functions, grouping them by assertion type.*" Table 6 reports the distribution of SQL functions across assertion types, revealing patterns—for example, `CAST` frequently co-occurs with logic bugs—suggesting recurring issues with type conversions. Future work could fully automate DBMS testing by leveraging LLM agents to automatically extract such insights through DSLs, implement new DSLs and reproduce them using **DBugScribe**.

DBugScribe DSL provides executable bug reproductions for further analysis in effective and reliable bug detection.

4.4.3 Non-Reproducible Report Triage. Beyond confirming genuine bugs, **DBugScribe** can automatically flag reports that are non-reproducible under the versions and environments stated in the report, helping prioritize follow-up and reduce unnecessary developer effort. By synthesizing executable scenarios from natural-language reports and attempting reproduction on specified DBMS versions, **DBugScribe** provides an automated pre-triage that surfaces reproduction discrepancies early. During our evaluation across bugs collected from recent testing tools (Section 4.1), **DBugScribe** identified an instance labeled "Confirmed" by tool authors while DBMS developers marked it "Cannot Reproduce".[6] **DBugScribe** did not reproduce this issue under the stated versions; we conducted manual verification attempts following the reported steps, which also failed to trigger the bug on the specified version. The case also shows promise in providing reproducible evaluation benchmarks for assessing DBMS testing tools.

DBugScribe flags non-reproducible or environment-dependent reports early via executable scenario synthesis.

4.4.4 Regression Detection. The unified, executable **DBugScribe** DSL enables systematic regression testing across DBMS versions. During our evaluation, we observed a case in which **DBugScribe** reproduced a MySQL bug both on the originally buggy version and on a more recent release[7]. The bug was found by EET [22], reported in 2023, and documented as fixed in MySQL 9.1.0. When we executed the synthesized **DBugScribe** DSL on the latest MySQL version (9.4.0), **DBugScribe** still reproduced the failure. We manually inspected the issue and confirmed the bug can be reproduced on 9.4.0 but not on 9.1.0, suggesting a regression after the documented fix. Although developers will likely integrate the reported bugs into the regression tests, this is usually a manual process and may be mistakenly forgotten.

DBugScribe identifies a regression bug in MySQL reintroduced after being marked fixed.

4.4.5 Cross-DBMS bug discovery. We evaluate the capability of **DBugScribe** 's synthesized DSL scenarios to be retargeted for uncovering bugs in other DBMSs.

[6]https://jira.mariadb.org/browse/MDEV-26405
[7]https://bugs.mysql.com/bug.php?id=112557

Table 7. Cross-DBMS bug reproduction by reusing executable **DBugScribe** DSL on the latest target DBMSs

Target DBMS	Total Tested	Reproduced	Passed	Error
MySQL	104	5	52	46
TiDB	103	10	44	49
MariaDB	111	22	68	21

For each bug reproduced in its original system, we retarget the DSL to the latest release of the other two systems (*e.g.*, run a MySQL-derived DSL on MariaDB and TiDB)—in this MySQL-family setting, this requires modifying only the `setup` section (`dbms_type`, `dbms_version`, `docker_image`) while leaving schema, operations, and assertions intact. We emphasize that *BugBench* includes only MySQL-compatible systems, and such "setup-only" bug reusing is not guaranteed across SQL dialect families (*e.g.*, from PostgreSQL to MySQL). Porting across dialects would require dialect-aware translation of SQL statements; we discuss the feasibility and associated challenges in Section 5.

Results. Table 7 reports the outcomes. Testing each DBMS with scenarios sourced from the other two successfully reproduced 37 additional bugs (5 sourced from MySQL, 10 from TiDB, and 22 from MariaDB). Notably, by reusing a MariaDB bug[8], we discovered a new crash bug in MySQL, which we reported to MySQL developers and received confirmation[9]. We did not submit all discovered logic bugs to vendors to reduce the risk of filing duplicates and to avoid overloading maintainers; instead, we prioritize releasing the executable cross-DBMS reproduction artifacts to facilitate follow-up and validation. For instance, the MariaDB bug #30255[10] is also reproducible on MySQL, which explicitly mentioned by MariaDB developers in comments.

Insights. The results show that bugs in one DBMS can often be reproduced in others, indicating shared vulnerabilities or similar implementation issues. We provided the following actionable insights. First, to make fair evaluation of the DBMS testing tools, we suggest researchers to select diverse DBMSs as targets, as many bugs can be ported across systems, and the evaluation evidence may be weakened. Fig. 4 shows that many recent DBMS testing tools target MySQL, MariaDB, and TiDB, while our findings suggest that one bug in one of these systems may also exist in the others. Second, DBMS developers can leverage **DBugScribe** to proactively test their systems against known bugs from other DBMSs, enhancing robustness and reliability, especially for emerging DBMSs that are built on and compatible with existing systems. Third, our approach can enhance knowledge sharing among DBMS developers by offering a unified framework for describing and reproducing bugs, supporting debugging and resolution.

DBugScribe discovers 37 additional bugs across DBMSs, including a new MySQL crash bug that we report and get confirmed.

5 Discussion

Potential extensions. We acknowledge that the current scope of **DBugScribe** is limited (Section 2.3), but the DSL and synthesis pipeline are general. Extending **DBugScribe** to specialized hardware or distributed deployments mainly requires corresponding environment management and execution support rather than changes to core semantics. **DBugScribe** can be adapted to handle

[8]https://jira.mariadb.org/browse/MDEV-34140
[9]https://bugs.mysql.com/bug.php?id=119157
[10]https://jira.mariadb.org/browse/MDEV-30255

these scenarios if initialization and configuration steps can be incorporated as executable DSL primitives, including deployment bootstrapping for distributed settings. Our current DSL and oracles target logic and crash bugs. To support performance-related issues, the DSL can specify execution settings (*e.g.*, repetitions and time budgets) and collect runtime metrics (*e.g.*, execution latency and query plans), with oracles expressed as runtime threshold or relative-slowdown predicates. Since performance is sensitive to environmental noise, robust reproduction requires statistically sound evaluation. The refinement loop can further incorporate runtime and query plan signals, in addition to errors and logical mismatches, to converge to stable and diagnostic reproductions.

Cross-dialect portability. Our current cross-DBMS bug discovery experiment (Section 4.4.5) is limited to MySQL-compatible systems, including MySQL, MariaDB, and TiDB, where modifying only the environment fields in `setup` allowed more than half of the bug-inducing cases to execute successfully without error. Reusing bug-inducing cases across different SQL dialect families is feasible but non-trivial. It requires rewriting the `sql_query` strings in `schema_data_steps` and `operations`, and validating or adapting oracles when semantics diverge. Key challenges include differences in supported statements, functions, operators, and type systems [67, 68]. However, reuse within a dialect family is often practical, since these systems typically share core semantics and differ mainly in dialect-specific features. For example, analogous to the MySQL family, **DBugScribe** can likely reuse bugs within the PostgreSQL family (*e.g.*, PostgreSQL, DuckDB, and CockroachDB) and execute them across systems.

Reproduction benefits. We identify the below actionable insights. First, **DBugScribe** can provide earlier, machine-checkable reproduction artifacts for new reports. Second, **DBugScribe** can support triage and confirmation workflows by providing ready-to-run scenarios that can be executed in the background and shared as concrete artifacts in issue threads. Moreover, the structured **DBugScribe** DSL representation provides a natural substrate for duplicate analysis (*e.g.*, clustering scenarios by structural similarity), which may support future duplicate detection and triage. Third, **DBugScribe** enables large-scale studies of bug characteristics and detection tools by automating reproduction.

6 Related Work

Database Bug Detection. Automated DBMS testing has progressed from random query generation [46] to techniques that leverage language-validity feedback [66] and stateful fuzzing [20] for crash detection. Logic bugs are commonly exposed via oracle-based testing, *e.g.*, PQS [43], TLP [42], and NoREC [41], with recent work further specializing in optimizer-centric settings [3, 5, 22, 48, 49, 51, 63]. Transactional correctness relies on graph-based oracles and isolation checking [9, 10, 21], while performance issues are targeted via regression and cardinality-estimation testing [4, 24, 33]. In contrast, **DBugScribe** focuses on reproducing already-reported bugs by extracting and synthesizing complete, executable scenarios from unstructured natural-language reports.

Database Bug Studies. Large-scale studies of DBMS bugs provide empirical guidance for testing and debugging. For example, Fu et al. [13] analyze built-in SQL function bugs, Wu et al. [54] study atomic DDL bugs under concurrent schema changes, and Cui et al. [10] characterize transaction bugs via write-specific serializability violations. These efforts emphasize characterization and detection, whereas **DBugScribe** translates unstructured reports into executable scenarios, enabling systematic validation and accumulating structured artifacts for downstream analysis.

Automated Bug Reproduction. Automated reproduction has been studied extensively in general software [6], with recent LLM-based systems such as LIBRO [25], AEGIS [53], and SWT-Bench [37]. Complementarily, bug report mining supports triage via metadata extraction [72]. DBMS bugs differ substantially due to stateful manifestation (schema and data), rich oracles, environment

and version sensitivity, and cross-DBMS portability needs. Moreover, effective reproduction often requires reconstructing DBMS-specific setup and configuration, and validating behaviors beyond functional outputs (e.g., crashes or error signatures). **DBugScribe** addresses these challenges with a DSL that has formal execution semantics, a hybrid synthesis pipeline combining LLM extraction with constraint validation, and containerized isolation for reproducible execution.

Data Generation from Workloads. Prior work on workload-driven data generation and regeneration seeks to approximate deployment-like behavior by synthesizing databases that satisfy constraints inferred from query workloads and query plans. *DataSynth* [2] formulates generation as constraint satisfaction and uses solvers to synthesize symbolic databases that satisfy cardinality constraints inferred from query plans. *Hydra* [44] targets "volumetric similarity" via dynamic regeneration, synthesizing large datasets whose intermediate operator cardinalities match those implied by client query plans without full materialization. *SAM* [57] uses supervised autoregressive models to generate databases that capture workload-relevant attribute correlations. In contrast, **DBugScribe** targets a different goal: reproducing concrete DBMS bug scenarios from natural-language reports by synthesizing executable artifacts that include schema/data, workload, environment, and validation oracles, rather than generating databases to match workload/plan statistics for benchmarking or workload modeling.

Text-to-SQL and Natural Language Interfaces. Text-to-SQL (NL2SQL) maps natural-language questions to executable SQL [34, 60]. Recent systems combine LLM-based semantic parsing [28, 45] with schema-aware encoders [52], achieving strong accuracy on Spider [60] and WikiSQL [69]. Generalization and robustness are improved via schema linking and retrieval-augmented prompting [50], execution-guided refinement [61], compositional parsing [47], sketch-based synthesis [12], and domain-tailored LLM frameworks [62], alongside reliability techniques such as adaptive abstention [7], automated validation/fixing [59], and schema naming assessment [35, 58]. **DBugScribe** applies NL2SQL-style extraction to DBMS testing: instead of answering a user query over a known schema, it extracts and refines multi-component reproduction scenarios (schema, data, workload, and oracles) from bug reports.

LLM for Database. Large language models have recently been explored for complex database management tasks beyond natural language interfaces, including system optimization and administration. LLM-R2 [30] and CrackSQL [70] leverage LLMs for query optimization and SQL dialect translation at the query level. GPTuner [26], λ-Tune [16], and Andromeda [8, 64] employ LLMs to extract tuning knowledge from manuals for parameter optimization and configuration debugging. These systems target routine optimization tasks with stable environments and known schemas. **DBugScribe** instead synthesizes complete bug reproduction scenarios from incomplete reports, requiring multi-component synthesis (*i.e.*, schema, data, queries, oracles) with execution-guided refinement beyond existing optimization-focused approaches.

7 Conclusion

We present **DBugScribe**, a system that advances the view that database bug reports can be treated as structured, executable artifacts by translating informal descriptions into structured scenarios with explicit environment, workload, and oracle semantics. Beyond enabling automated reproduction in a controlled setting, these DSL-based representations provide reusable building blocks for downstream reliability tasks, including regression checking, cross-system retargeting within compatible families, and systematic analyses of recurring bug patterns. Overall, this direction opens opportunities to build principled, reproducible benchmarks and to integrate structured reproduction artifacts into DBMS maintenance workflows, facilitating more transparent evaluation, faster iteration across systems, and tighter feedback loops from reporting to validation.

References

[1] Abdulaziz Alaboudi and Thomas D. LaToza. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empir. Softw. Eng.* 28, 5 (2023), 117.

[2] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 685–696.

[3] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2060–2071.

[4] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 133, 13 pages.

[5] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3, Article 188 (May 2024), 26 pages.

[6] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 257–269.

[7] Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2025. Reliable Text-to-SQL with Adaptive Abstention. *Proc. ACM Manag. Data* 3, 1 (2025), 69:1–69:30.

[8] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *Proc. ACM Manag. Data* 3, 1, Article 13 (Feb. 2025), 27 pages.

[9] Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 754–768.

[10] Ziyu Cui, Wensheng Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. 2025. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. *Proc. VLDB Endow.* 18, 8 (Sept. 2025), 2547–2560.

[11] Wenjing Deng, Qiuyang Mang, Chengyu Zhang, and Manuel Rigger. 2024. Finding Logic Bugs in Spatial Database Engines via Affine Equivalent Inputs. *Proc. ACM Manag. Data* 2, 6, Article 235 (Dec. 2024), 26 pages.

[12] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1534–1547.

[13] Jingzhou Fu, Jie Liang, Zhiyong Wu, Yanyang Zhao, Shanshan Li, and Yu Jiang. 2025. Understanding and Detecting SQL Function Bugs: Using Simple Boundary Arguments to Trigger Hundreds of DBMS Bugs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1061–1076.

[14] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1132–1145.

[15] Xiyue Gao, Zhuang Liu, Yiran Shen, Hui Li, Yingfan Liu, Hongjun Xiao, Yanguo Peng, and Jiangtao Cui. 2025. Fucci: Database Transaction Fuzzing via Random Conflict Construction and Multilevel Constraint Solving. *Proc. VLDB Endow.* 18, 6 (Aug. 2025), 1879–1891.

[16] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1, Article 2 (Feb. 2025), 26 pages.

[17] Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. SQLens: An End-to-End Framework for Error Detection and Correction in Text-to-SQL. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[18] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.

[19] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. *IEEE Transactions on Knowledge and Data Engineering* 37, 12 (2025), 7328–7345.

[20] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4949–4965.

[21] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 397–417.

[22] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 821–835.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 77–88.

[24] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70.

[25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2312–2323.

[26] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (April 2024), 1939–1952.

[27] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1509–1520.

[28] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 13067–13075.

[29] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1835, 28 pages.

[30] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *Proc. VLDB Endow.* 18, 1 (2024), 53–65.

[31] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 668–681.

[32] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. WingFuzz: Implementing Continuous Fuzzing for DBMSs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 479–492.

[33] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems Using Equivalent Queries. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 225–236.

[34] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Natural Language to SQL: State of the Art and Open Problems. *Proc. VLDB Endow.* 18, 12 (2025), 5466–5471.

[35] Kyle Luoma and Arun Kumar. 2025. SNAILS: Schema Naming Assessments for Improved LLM-Based SQL Inference. *Proc. ACM Manag. Data* 3, 1 (2025), 77:1–77:26.

[36] Qiuyang Mang, Jinsheng Ba, Pinjia He, and Manuel Rigger. 2025. Finding Logic Bugs in Graph-processing Systems via Graph-cutting. *Proc. ACM Manag. Data* 3, 3, Article 163 (June 2025), 27 pages.

[37] Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. SWT-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems* 37 (2024), 81857–81887.

[38] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. ACM, 199–209.

[39] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. 2022. Works for Me! Cannot Reproduce - A Large Scale Empirical Study of Non-reproducible Bugs. *Empir. Softw. Eng.* 27, 5 (2022), 111.

[40] Shanto Rahman, Saikat Dutta, and August Shi. 2025. Understanding and Improving Flaky Test Classification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 320 (Oct. 2025), 27 pages.

[41] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152.

[42] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages.

[43] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

[44] Anupam Sanghi, Raghav Sood, Jayant Haritsa, and Srikanta Tirthapura. 2018. Scalable and dynamic regeneration of big data volumes. (2018).

[45] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 9895–9901.

[46] Andreas Seltenreich. 2022. Sqlsmith. https://github.com/anse1/sqlsmith

[47] Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional Generalization and Natural Language Variation: Can a Semantic Parsing Approach Handle Both?. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*. ACL, 922–938.

[48] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proc. VLDB Endow.* 17, 8 (may 2024), 1884–1897.

[49] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. *Proc. VLDB Endow.* 18, 7 (Aug. 2025), 2281–2294.

[50] Ruoxi Sun, Sercan Ö. Arik, Alexandre Muzio, Lesly Miculicich, Satya Kesav Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. 2024. SQL-PaLM: Improved large language model adaptation for Text-to-SQL. *Trans. Mach. Learn. Res.* 2024.

[51] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1, Article 55 (may 2023), 26 pages.

[52] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 7567–7578.

[53] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2025. AEGIS: An Agent-based Framework for Bug Reproduction from Issue Descriptions. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 331–342.

[54] Zhiyong Wu, Jie Liang, Jingzhou Fu, Wenqian Deng, and Yu Jiang. 2025. DDLumos: Understanding and Detecting Atomic DDL Bugs in DBMSs. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1327–1341.

[55] Zhiyong Wu, Jie Liang, Jingzhou Fu, Wenqian Deng, and Yu Jiang. 2025. Fawkes: Finding Data Durability Bugs in DBMSs via Recovered Data State Verification. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 670–684.

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

[57] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1542–1555.

[58] Xiao Yang, Mo Sha, Yiran Li, Suyang Zhong, Sheng Wang, Fangyuan Zhou, and Feifei Li. 2026. SQLens: Continuous Code-to-SQL Visibility in the Wild. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data* (Bengaluru, India) *(SIGMOD '26)*.

[59] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data* 3, 3 (2025), 134:1–134:28.

[60] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 3911–3921.

[61] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. SParC: Cross-Domain Semantic Parsing in Context. In *Proceedings of the 57th*

Annual Meeting of the Association for Computational Linguistics (ACL). ACL, 4511–4523.

[62] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 93–105.

[63] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data* 3, 1, Article 24 (Feb. 2025), 24 pages.

[64] Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. 2026. Why Database Manuals Are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents. *Proc. VLDB Endow.* 19, 6 (2026), 1358–1371.

[65] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.

[66] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970.

[67] Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proc. ACM Manag. Data* 2, 6, Article 253 (Dec. 2024), 26 pages.

[68] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1677–1692.

[69] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103* (2017).

[70] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *Proc. ACM Manag. Data* 3, 3, Article 141 (June 2025), 26 pages.

[71] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 836–848.

[72] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Software Eng.* 36, 5 (2010), 618–643.

Received October 2025; revised January 2026; accepted February 2026