HarnessDB: Evolving Differential-Augmented Harnesses for Logical Bug Detection in Database Engines

Anonymous ACL submission

Abstract

Metamorphic-oracle logical testing (MOLT) is effective in detecting logic bugs in modern Database Management Systems (DBMSs), but many low-resource DBMSs lack sufficient test data. Recent LLM-based methods address this problem by migrating MOLT cases from popular DBMSs through translation and mutation. However, they still lack effective constraints over the migration process, which may cause test logic to drift from the intended oracle and makes it difficult for LLMs to avoid recurring error patterns. To address these limitations, we propose **HarnessDB**, an LLM-powered framework for more reliable and efficient MOLT migration. First, HarnessDB constructs differential knowledge from failed and corrected cases, allowing common error patterns to be explicitly captured and reused. Second, it continuously optimizes Harness Memory as constraints, enabling the LLM to avoid repeated mistakes and preserve the intended oracle logic during later translation and mutation. Experiments on eight DBMSs show that HarnessDB consistently outperforms the state-of-the-art baselines, reduces token consumption, and uncovers multiple previously unknown logic bugs confirmed by developers.

1 Introduction

Modern software systems critically depend on the reliability of Database Management Systems (DBMSs) (Zhou et al., 2020). Logic bugs in DBMSs are especially harmful because they may return incorrect results without crashes or warnings, potentially causing serious problems (Segura et al., 2016; Chen et al., 2018). Metamorphic-oracle based logical bug detection techniques (MOLTs) (Chen et al., 1998) have proven the most effective approach for uncovering such defects, with representative methods such as NoREC (Rigger and Su, 2020a), TLP (Rigger and Su, 2020b), DQE (Song et al., 2023), and Pinolo (Hao et al., 2023). Nevertheless, existing MOLT implementations rely heavily on hand-written, DBMS-specific adaptation code, where SQLRight (Liang et al., 2022) alone adds about 90,000 lines solely to support PostgreSQL. Such high adaptation cost confines mature MOLT support to a few popular DBMSs, leaving many of the 431+ active DBMSs[1] on the market without effective logical-bug detection.

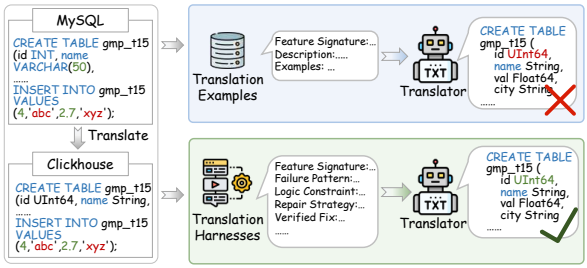

Figure 1: When only translation examples are retrieved, the LLM may still introduce dialect-specific errors and fail to preserve the intended testing logic. By contrast, harness-guided translation reuses differential experience from previous failed and corrected cases, to explicitly constrain the translation process and avoid recurring cross-dialect errors.

Large Language Models (LLMs) offer a promising direction for reducing this adaptation cost. QTRAN (Lin et al., 2025) is a representative method that decomposes MOLT migration into a translation phase, which uses retrieval-augmented prompts to convert SQL queries across dialects, and a mutation phase, which applies oracle-specific rules to generate additional test cases. This approach substantially reduces the manual effort of adapting MOLTs to new DBMSs.

However, QTRAN provides limited constraints over the migration process and faces two key problems, as illustrated in Figure 1. First, when a translation fails, it mainly relies on the feedback to repair the problem, without explicitly constrain-

[1] https://db-engines.com/en/ranking

ing the intended testing logic. Repeated retries not only increase token and time costs, but may also cause the generated queries to drift from the original oracle, weakening their ability to expose logic bugs on the target DBMS. Second, its knowledge base is built mainly from official documentation and positive references, while ignoring negative migration experience from failed cases. As a result, the LLM may repeatedly encounter similar cross-dialect errors without learning how to avoid them, further amplifying retry overhead and testing-logic drift.

Harness engineering provides a useful perspective for addressing these limitations. It is an emerging practice in AI-assisted software development that builds reusable scaffolds around LLMs, such as verified patterns, task-specific constraints, and domain knowledge (Li et al., 2026). Instead of asking the LLM to solve each task from scratch, a harness guides the model with accumulated experience and explicit constraints, making its behavior more reliable and controllable.

Inspired by harness engineering, we propose HarnessDB, an LLM-powered framework for reliable and efficient MOLT migration. HarnessDB turns migration experience into evolving constraints through two key designs. First, it constructs differential-augmented knowledge from failed migration cases and their corrected versions, allowing recurring syntactic and semantic error patterns across SQL dialects to be captured as negative experience. Second, it abstracts successful translation and mutation cases into Harness Memory, which provides reusable constraints for later migrations. Together, these designs help the LLM avoid repeated mistakes, reduce redundant migration attempts, and preserve the intended oracle logic for subsequent testing.

Figure 1 highlights the effectiveness of HarnessDB compared with the QTRAN's migration paradigm. By incorporating differential-augmented knowledge and evolving Harness Memory, HarnessDB provides stronger migration constraints than documentation-based retrieval and error-feedback retries. As a result, it reduces redundant LLM calls, better preserves the intended oracle logic, and generates migrated test cases that more reliably support logic-bug detection. Our evaluation across eight DBMSs and four MOLTs shows that HarnessDB achieves higher migration accuracy than the state-of-the-art baseline and uncovers 24 previously unknown logic bugs, 16 of which have been confirmed by developers. The contributions are summarized as follows:

- **Harness Engineering for MOLT Migration.** We introduce harness engineering into LLM-based MOLT migration, using reusable constraints to guide translation and mutation.
- **HarnessDB Framework.** We propose HarnessDB, a framework that combines differential-augmented knowledge and evolving Harness Memory to reduce repeated errors and preserve oracle logic.
- **Comprehensive Evaluation.** We evaluate HarnessDB across eight DBMSs and four MOLTs, showing its effectiveness in improving MOLT migration and supporting logic-bug detection.

2 Related Work

2.1 Logic Bug Detection in DBMSs

Existing logic bug detection techniques mainly rely on three families of oracles (Segura et al., 2016; Chen et al., 2018). Differential oracles compare query results across DBMSs (Cui et al., 2022; Slutz, 1998; Lemieux et al., 2022), but they may suffer from false positives caused by dialect-specific semantics. Oracle-guided synthesis methods, such as PQS (Rigger and Su, 2020d), construct queries that must return specific rows but require tailored logic for each DBMS. Metamorphic oracles, especially MOLTs, have been particularly successful: NoREC (Rigger and Su, 2020a), TLP (Rigger and Su, 2020b), DQE (Song et al., 2023), and subsequent variants (Hao et al., 2023; Ba and Rigger, 2023a,b; Song et al., 2024; Wang et al., 2025) apply mutation rules that define expected set relationships, collectively uncovering hundreds of bugs. SQLancer (Rigger and Su, 2020c) unifies many MOLTs in an integrated framework, while intramorphic testing (Rigger and Su, 2022) offers an alternative paradigm.

2.2 LLM-Driven Fuzzing and SQL Migration

2.3 Task Formulation

LLMs have promoted a new generation of fuzzing tools (Jiang et al., 2024). Recent studies apply LLMs to fuzzing across domains, including general-purpose fuzzing (Xia et al., 2024), deep-learning libraries (Deng et al., 2023), and bug reproduction (Kang et al., 2023). In databases,

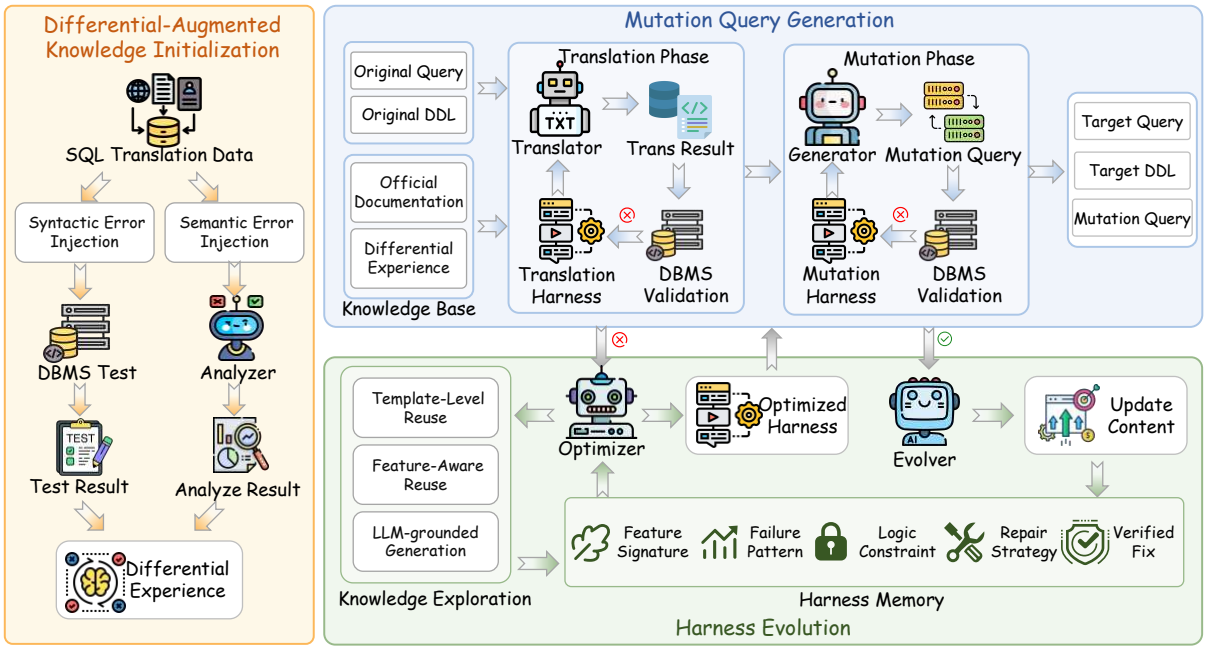

Figure 2: Overall framework of HarnessDB.

Sedar (Fu et al., 2024) and QTRAN (Lin et al., 2025) pioneer LLM-driven cross-DBMS SQL migration. Broader LLM-based test generation (Dakhel et al., 2024; Ibrahimzada et al., 2023) and advances in LLM reasoning (Wei et al., 2022; Chowdhery et al., 2023; Cobbe et al., 2021) further expand these capabilities. Code-intelligence models (Feng et al., 2020; Wang et al., 2021; Nijkamp et al., 2023; Li et al., 2023; Rozière et al., 2023; Guo et al., 2024), unsupervised code translation (Lachaux et al., 2020), and dialect-agnostic SQL simplification (Lin et al., 2024) also share the goal of reducing manual cross-dialect porting effort. Prompt optimization (Khattab et al., 2023; Zhou et al., 2023; Jiang et al., 2023) and structured reasoning paradigms (Yao et al., 2023a,b; Wang et al., 2023) inform our prompt design and memory reuse strategies. However, existing methods mainly rely on positive references and treat the LLM as stateless, while LLM hallucination (Huang et al., 2025) further compounds these issues. HarnessDB addresses both limitations by incorporating negative differential experience and persistent Harness Memory.

3 Methodology

We propose HarnessDB, an evolving framework designed to migrate existing MOLT for logical bug detection in database engines which are not support by these MOLTs. The framework of HarnessDB is illustrated in Figure 2. Similar to QTRAN (Lin et al., 2025), HarnessDB consists of a translation phase and a mutation phase, which is enriched with a **differential-augmented knowledge base** (Section 3.1) during translation phase and runs under an **evolving harness** (Section 3.2) across both phases. After these two phases, HarnessDB will execute the correctly migrated SQL in the target database and retain error logs for manual review.

Let $\mathcal{S}$ and $\mathcal{T}$ denote the source database and target database, Q denote the query statement used to verify the results and R denote the mutation rule applied to derive the mutated statement Q'. An MOLT test case is a tuple $(\mathrm{DDL}_s, Q_s, Q'_s, R)$ from $\mathcal{S}$. The migration task produces $(\mathrm{DDL}_t, Q_t, Q'_t, R)$ for $\mathcal{T}$ via two phases: a **translation phase** f_{trans} that converts (DDL_s, Q_s) to target-dialect SQL aided by a translation knowledge base $\mathcal{K} = (\mathcal{D}, \mathcal{E}, \mathcal{M})$ (official documentation, differential experience, feature mapping), and a **mutation phase** f_{mut} that converts the correctly translated query Q_t to Q'_t by applying mutation rule R under the supervision of a fine-tuned LLM. Spanning both phases, the **harness** $\mathcal{H} = (\mathcal{H}_{\mathrm{trans}}, \mathcal{H}_{\mathrm{mut}})$ stores successful operation patterns and reuses them at two tiers: exact template matches bypass the LLM, while partial matches serve as few-shot context. Differential experience $\mathcal{E}$ is consumed only by f_{trans}, since

dialect-specific pitfalls are intrinsic to translation; the harness, in contrast, applies uniformly to both phases. This design mirrors the two-phase decomposition established in prior cross-dialect SQL migration work (Lin et al., 2025; Fu et al., 2024; Lin et al., 2024).

3.1 Differential-Augmented Knowledge Base

Knowledge Base Construction. We select data types, functions, and operators as the three categories of mapping features following QTRAN (Lin et al., 2025), and update the crawling pipeline to capture standard and dialect-specific features from the latest official documentation of all eight target DBMSs. The dialect proportion exceeds 10% on average, which is crucial in the SQL translation process where even minor differences can lead to errors. Full statistics, crawling details, and our retrieval-augmented generation pipeline (Lewis et al., 2020), which indexes feature-level knowledge for efficient look-up during translation can be found in Appendix A.

Differential Experience Generation. Except correct reference, human also learn from past mistakes (Mnih et al., 2015; Schaul et al., 2016). To equip the translation LLM with analogous error-avoidance awareness, we construct for each feature a correct target translation and then synthesize two categories of deliberately erroneous translations: (1) *syntactic errors*, i.e., subtle faults that cause guaranteed execution failure (e.g., missing commas, mismatched parentheses, or slight keyword typos), together with the captured DBMS error descriptions; (2) *semantic errors*, i.e., syntactically valid but logically flawed translations that execute without crashing yet return incorrect results (e.g., using a wrong join type, a boundary condition error such as > instead of >=, or logical operator confusion), accompanied by concise naturallanguage analyses of the injected flaw. Detailed prompt template to generate differential experience can be found in Appendix C.

Figure 3 applies PCA (Maćkiewicz and Ratajczak, 1993) to the embeddings of $\mathcal{D}$ and $\mathcal{E}$ to verify that differential experience genuinely enriches the prompt's information rather than merely duplicating documentation. Official documentation samples (blue) form a tight cluster, exhibiting low diversity, while differential experience samples (orange) are widely scattered, confirming complementary feature coverage that fills in edge-case regions of the prompt space.

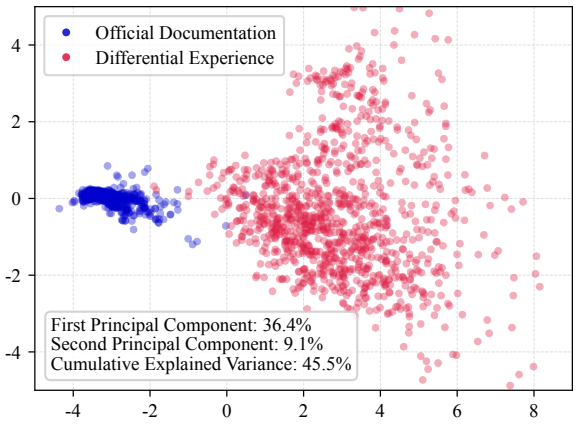

Figure 3: PCA visualization of official documentation (blue, clustered) versus differential experience (orange, widely scattered), demonstrating the complementary feature coverage provided by negative knowledge.

Feature Aggregation and Restoration. To reduce feature lookup to character-level token matching at retrieval time, we pre-aggregate semantically equivalent features across DBMSs into a unified mapping $\mathcal{M}$ and run a low-hallucination restoration pass that converges within 3–4 rounds. The two underlying algorithms and the convergence curve are detailed in Appendix B.

3.2 Evolving Harness for MOLT Migration

Harness Architecture. The harness $\mathcal{H} = (\mathcal{H}_{\text{trans}}, \mathcal{H}_{\text{mut}})$ comprises two stores with an identical record schema and access protocol. Each record is a triple (φ, τ, σ), where φ is a *feature signature*, the set of data types, functions, and operators tokenized from the SQL (e.g., `INT`, `SUM`, `=`); τ is a *structural template*, an AST-derived signature that abstracts identifiers and literals into type tags while preserving the clause skeleton (e.g., `Select(FROM,WHERE,GROUP,...)`); and σ is the *concrete payload*, the verified target-DBMS SQL plus the source SQL for few-shot formatting. $\mathcal{H}_{\text{trans}}$ is keyed by $(\mathcal{S}, \mathcal{T})$ and matches on exact τ or Jaccard overlap over φ; $\mathcal{H}_{\text{mut}}$ is keyed by MOLT and additionally filters by mutation rule R. Both stores are persisted as thread-safe JSONL logs and share a unified generation and reuse procedure. The proposed method is outlined in Algorithm 1.

Memory Generation. Upon every successful LLM operation, the system commit the (φ, τ, σ) record to the appropriate store. Records sharing an identical τ (and R, for mutation) are overwritten with the most recent entry, bounding growth

by template diversity.

Memory Reuse with Prompt Optimization. For an incoming SQL X, the system extracts (φ, τ) and queries the harness. At *Tier 1* (exact τ match), the stored payload σ is returned directly, bypassing the task LLM and incurring zero tokens. The result is still verified; if it fails, the system falls through to lower tiers. At *Tier 2* (Jaccard similarity ≥ 0.5, top-$k = 3$), the most similar records are formatted as few-shot demonstrations and injected *in place of* the full documentation block, while differential experience is retained for error avoidance—replacing 2,000–4,000 tokens of documentation with 200–400 tokens of compact exemplars. Before reaching the task LLM, a lightweight prompt-optimization pass (GPT-5 mini) consolidates redundant clauses. At *Tier 3* (miss), the full knowledge base is used without optimization. Regardless of tier, the system retries with DBMS error feedback appended to the prompt upon failure. Once verified, the new result is recorded, enriching the store for future invocations.

Algorithm 1 Three-Tier Evolving Harness for MOLT Migration

Require: SQL input X, harness $\mathcal{H}$, stage-specific key K, threshold θ, max retries N
Ensure: Verified payload Y

```
(φ, τ) ← Extract(X)
C ← H.Lookup(K, φ, τ)
if C has exact τ match then
    /* Tier 1: skip task LLM */
    Y ← C.σ
    if Verify(Y) then return Y
    end if
end if
P ← BuildPrompt(X)
if Jaccard(C.φ, φ) ≥ θ then
    /* Tier 2: replace docs with few-shot, retain diff */
    InjectFewShot(P, C.top-k)
    InjectDiff(P, K_diff)
    P ← OptimizePrompt(P)
else
    /* Tier 3: full knowledge base */
    InjectDocs(P, K_docs)
    InjectDiff(P, K_diff)
end if
for i ← 1 to N do
    Y ← TaskLLM(P)
    if Verify(Y) then
        H.Record(K, φ, τ, Y)
        return Y
    end if
    P ← AppendErrorFeedback(P, Y)
end for
return ⊥
```

4 Experimental Setup

All experiments are conducted on a server equipped with dual Intel Xeon Gold 6530 CPU and a single RTX 5090 GPU. To probe how each method behaves with translation backbones of different capacity, we evaluate three translation LLMs: GPT-4o mini, GPT-5 mini, and Qwen3.5-9B (Qwen Team, 2025), and use a single Supervised Fine-Tuned Qwen3.5-9B as the shared mutation LLM throughout the main experiments. For the fine-tuning, we adopt LoRA (Hu et al., 2022) with rank $r = 8$ and $\alpha = 16$. Refer to Appendix D for detailed configuration.

4.1 Baseline

We compare HarnessDB against QTRAN (Lin et al., 2025), the current state-of-the-art LLM-driven MOLT migration method. QTRAN employs a two-stage translation-mutation pipeline with official documentation as the sole external knowledge source. HarnessDB additionally injects differential experience into the translation knowledge base and activates the evolving harness in both phases.

4.2 Dataset

For fair comparison, we collect data from four representative MOLTs: NoREC, TLP, DQE, and Pinolo, each running on the DBMS where it achieves the highest bug-finding effectiveness: SQLite for NoREC, MySQL for TLP and Pinolo, and TiDB for DQE. For NoREC, TLP, and DQE, we collect 200 mutation samples each. Pinolo, which we observe to expose 18 distinct mutation strategies in its source code, is sampled at 10 mutation query pairs per strategy, yielding 180 samples in total.

5 Experimental Analysis

To evaluate the effectiveness of HarnessDB, we conduct experiments from four perspectives and organize this section around the following research questions **(RQs)**:

RQ1: How effective is HarnessDB in migrating MOLT test cases across heterogeneous DBMSs compared with the state-of-the-art LLM-based migration method?

RQ2: Can HarnessDB improve migration efficiency by reducing token consumption while maintaining strong migration performance?

Table 1: Overall migration score (Defined in Equation 1) across translation backbones and DBMSs. The mutation backbone is fixed to the LoRA-fine-tuned Qwen3.5-9B for all rows.

Model	ClickHouse	DuckDB	MariaDB	MonetDB	MySQL	PostgreSQL	SQLite	TiDB	**Avg.**
QTRAN (GPT-4o mini)	2.5	36.6	95.3	20.8	28.8	30.6	81.9	21.3	39.7
QTRAN (GPT-5 mini)	23.2	41.3	96.5	32.6	31.3	46.6	84.7	30.5	48.3
QTRAN (Qwen3.5-9B)	17.3	26.9	10.8	9.4	9.0	27.2	51.9	8.5	20.1
HarnessDB (GPT-4o mini)	10.3	48.4	100	36.4	33.5	33.9	65.0	33.8	**45.2**
HarnessDB (GPT-5 mini)	27.2	51.8	100	36.8	35.5	51.5	88.2	34.8	**53.2**
HarnessDB (Qwen3.5-9B)	24.8	34.5	18.5	16.8	16.5	35.0	59.0	16.0	**27.6**

RQ3: How much does each key component of HarnessDB contribute to translation and mutation correctness?

RQ4: How does HarnessDB guide an end-to-end MOLT migration case through harness-guided translation and mutation?

5.1 Main Results

Logic Bug Detection. We deploy HarnessDB in an autonomous testing campaign across all eight target DBMSs for five hours each. HarnessDB uncovered multiple previously unknown logic bugs, including incorrect query optimization, mishandled NULL propagation, type coercion errors, and aggregate function miscalculations. Several bugs have been confirmed and are under developer review; we are awaiting community feedback on the remaining reports. A detailed bug list is maintained in the project repository. These results confirm that HarnessDB-generated test cases are practically effective at surfacing genuine logic defects across heterogeneous production DBMSs.

Migration Success and Resource Consumption. We define the migration score as a weighted combination of the translation success rate and the mutation pass rate, with a balancing parameter λ:

$$\text{migration score} = \lambda \cdot \frac{\text{Translation Valid}}{\text{Total}} + (1-\lambda) \cdot \frac{\text{Mutation Valid}}{\text{Total}}. \quad (1)$$

To treat both stages equally, we set $\lambda = 0.5$, so the score reduces to the arithmetic mean of the two validity rates. This composite metric captures the overall two-stage effectiveness, isolating the end-to-end carrythrough quality that our two innovations target.

Table 1 reports the migration score (mean of translation and mutation validity rates) across all eight target DBMSs under three translation backbones. HarnessDB consistently outperforms QTRAN on every (LLM, DBMS) cell, with average gains of +5.5 (GPT-4o mini), +4.9 (GPT-5 mini), and +7.5 (Qwen3.5-9B). Two patterns emerge. First, gains exhibit diminishing returns with model capacity: the weakest model benefits most as differential experience supplies dialect knowledge it cannot infer from documentation alone. Second, gains concentrate on dialect-sensitive DBMSs (e.g., MonetDB +7.4, MariaDB +7.7, PostgreSQL +7.8 under Qwen3.5-9B), while near-saturation DBMSs show narrower gaps. The cross-model consistency confirms that the improvement stems from the framework rather than from any particular LLM.

Figure 4 plots per-test-case API calls and token consumption. HarnessDB reduces token expenditure by bypassing the LLM at Tier 1 and replacing full documentation with compact few-shot at Tier 2, while API calls rise slightly due to lightweight feature extraction and harness lookup. The net effect favors token reduction, which dominates LLM inference cost.

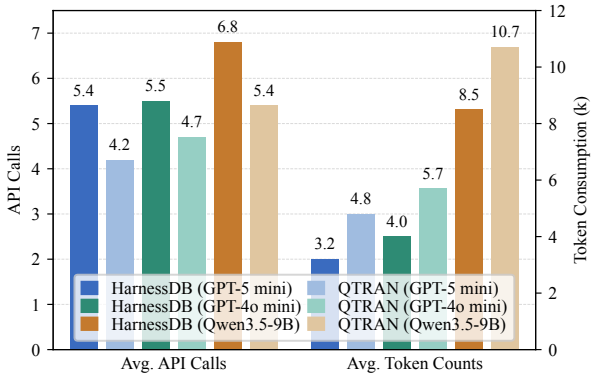

Figure 4: Per-test-case API call count and token consumption for HarnessDB vs. QTRAN.

5.2 Ablation Study

We conduct two ablation studies to isolate the contributions of each component.

Translation Phase Ablation. Table 2 re-

Table 2: Ablation study of the translation phase. Syntactic correctness rate (Syn.) and semantic correctness rate (Sem.) under four configurations, evaluated on three translation models.

Model	Metric	Method	ClickHouse	DuckDB	MariaDB	MonetDB	MySQL	PostgreSQL	SQLite	TiDB
GPT-4o-mini	Syn.	w/o Docs	69.3	83.2	100	82.1	90.1	91.5	100	83.4
		w/o Diff	71.8	82.5	100	81.4	89.2	90.8	100	86.2
		w/o Harness	72.6	84.1	100	83.7	91.0	92.0	100	86.5
		HarnessDB	**73.4**	**85.7**	100	**84.1**	**93.2**	**93.8**	100	**87.8**
	Sem.	w/o Docs	36.1	47.3	80.2	47.1	54.2	52.1	90.3	53.2
		w/o Diff	34.8	46.7	79.5	46.3	53.8	51.3	89.8	49.4
		w/o Harness	35.9	47.8	81.0	46.8	55.1	52.6	91.1	52.8
		HarnessDB	**37.0**	**48.9**	**82.5**	46.9	**56.8**	**53.9**	**91.6**	53.4
GPT-5-mini	Syn.	w/o Docs	71.5	85.4	100	84.3	92.5	93.4	100	85.6
		w/o Diff	73.2	84.4	100	83.2	91.6	92.6	100	88.4
		w/o Harness	74.1	86.1	100	85.9	93.4	94.2	100	88.6
		HarnessDB	**75.1**	**87.1**	100	**85.9**	**95.4**	**95.2**	100	**89.6**
	Sem.	w/o Docs	38.4	49.5	82.4	49.4	56.8	54.3	92.6	55.6
		w/o Diff	36.9	48.9	81.2	48.7	56.4	53.8	91.5	51.3
		w/o Harness	38.3	49.9	83.2	48.6	57.8	54.8	92.9	54.4
		HarnessDB	**39.3**	**50.9**	**84.0**	48.6	**59.0**	**55.1**	**93.3**	55.4
Qwen3.5-9B	Syn.	w/o Docs	62.8	76.3	98.5	74.6	83.1	82.7	98.9	76.2
		w/o Diff	64.5	77.8	99.1	75.3	84.0	83.5	99.0	79.6
		w/o Harness	65.3	78.9	99.2	76.8	85.2	84.3	99.0	80.1
		HarnessDB	**66.8**	**79.7**	**99.3**	**77.4**	**86.9**	**85.8**	99.0	**81.9**
	Sem.	w/o Docs	32.5	42.1	74.6	42.8	48.9	46.5	86.7	48.3
		w/o Diff	31.2	41.6	73.2	41.9	48.1	45.2	85.9	44.6
		w/o Harness	32.1	42.5	74.9	42.3	49.3	46.8	87.2	47.9
		HarnessDB	**33.0**	**43.7**	**76.1**	42.4	**50.6**	**48.2**	**87.8**	48.5

ports syntactic (Syn) and semantic (Sem) correctness under four configurations: *w/o Docs.*, *w/o Diff.*, *w/o Harness*, and the full *HarnessDB*. Three findings emerge. First, HarnessDB achieves the highest Syn and Sem across all models and DBMSs. Second, differential experience independently drives semantic correctness: *w/o Docs* (differential only) outperforms *w/o Diff* (documentation only) on Sem for most DBMSs, confirming that negative exemplars instill error awareness that positive documentation alone does not convey. Third, the harness provides a consistent gain of 1–2 points, acting as a general-purpose reliability layer.

Mutation Phase Ablation. Table 3 reports correct mutation rates for three configurations: *w/o Fine-tuning* (base model + harness), *w/o Harness* (SFT model only), and full *HarnessDB*. Fine-tuning is indispensable: removing it causes a collapse of 37–53 points across all four MOLTs, sharpest on DQE (–53.0, from 85.0 to 32.0). The harness adds a modest but consistent gain (+1–3 points), largest on DQE where stored mutation patterns provide the most value for complex multi-statement rules. The two components are complementary: fine-tuning instills rule-conformant rewriting, while the harness sharpens choices on structurally complex cases.

Table 3: Ablation study of the mutation phase. Correct mutation rate across four MOLTs under three configurations, evaluated on queries correctly translated by Qwen3.5-9B with the fine-tuned Qwen3.5-9B as the mutator.

Configuration	NoREC	TLP	Pinolo	DQE
w/o Fine-tuning	46.0	57.0	39.0	32.0
w/o Harness	89.0	93.0	87.0	82.0
HarnessDB	**91.0**	**94.0**	**88.0**	**85.0**

5.3 Case Study

Figure 5 illustrates HarnessDB's end-to-end pipeline through a concrete Pinolo test case migrated from MySQL to ClickHouse. The example uses the `FixMDistinctU` strategy, which removes the `DISTINCT` keyword—broadening the result set so that any row missing from the original output signals a logic bug. The source DDL mixes MySQL-specific patterns: inline `KEY(id)`, the `DOUBLE` type, and `TEXT`. The differential KB guides the translation: `INT`→`UInt64`, `VARCHAR(50)`→`String`, `DOUBLE`→`Float64`, `TEXT`→`String`, and the inline `KEY` clause migrates

Translation Phase

```mysql
-- DDL
CREATE TABLE gmp_t15 (id INT, name VARCHAR(50), val DOUBLE, city TEXT, KEY(id));
INSERT INTO gmp_t15 VALUES (4,'abc',2.7,'xyz');
-- Query
SELECT DISTINCT id, name, val, city FROM gmp_t15;
```

Knowledge base map + Harness:
INT -> UInt64, VARCHAR(50)/TEXT -> String, DOUBLE -> Float64, KEY(id) -> ENGINE = Merge Tree() ORDER BY id

```clickhouse
-- Translated DDL
CREATE TABLE gmp_t15 (id UInt64, name String, val Float64, city String) ENGINE = MergeTree() ORDER BY id;
INSERT INTO gmp_t15 VALUES (4,'abc',2.7,'xyz');
-- Translated Query
SELECT DISTINCT id, name, val, city FROM gmp_t15;
```

Mutation Phase

```clickhouse
-- Translated Query
SELECT DISTINCT id, name, val, city FROM gmp_t15;
```

Fine-tuned Qwen3.5-9B + Harness: applies FixMDistinctU (DISTINCT -> removed):

```clickhouse
-- Mutated Query
SELECT id, name, val, city FROM gmp_t15;
```

Oracle: the DISTINCT result must be a subset ofthe non-DISTINCT result; any row in the original outputmissing from the mutated output indicates a logic bug.

Figure 5: Case study of HarnessDB migrating a Pinolo (`FixMDistinctU`) test case from MySQL to ClickHouse. The differential KB drives dialect-specific type mappings and engine syntax; the harness caches the pattern for future reuse.

to ClickHouse's `ENGINE = MergeTree() ORDER BY id`. The query body (`SELECT DISTINCT ...`) passes through unchanged, since both dialects share the same DML syntax. The harness records this successful pattern, so future MySQL-to-ClickHouse migrations with similar type mappings can reuse it at Tier 1 (zero LLM tokens) or Tier 2 (compact few-shot).

6 Conclusions

In this work, we propose HarnessDB, a novel LLM-powered framework that addresses two fundamental limitations in existing MOLT migration approaches: the lack of error-awareness during migration and the drift of generated queries from the original oracle under repeated retries. HarnessDB combines a differential-augmented knowledge base that injects structured negative experience to expose recurring cross-dialect errors, and a reusable memory harness that records successful translation and mutation patterns to preserve the intended oracle logic and suppress redundant retries. Across eight mainstream DBMSs and four MOLTs, HarnessDB improves migration accuracy while substantially reducing per-translation token consumption, and uncovers 24 previously unknown logic bugs, 16 of which have been confirmed by developers. This work opens a promising direction for making LLM-driven software engineering tools simultaneously more reliable and more cost-efficient, with potential applications beyond database testing to broader code translation (Lachaux et al., 2020; Feng et al., 2020) and test generation (Dakhel et al., 2024; Kang et al., 2023) tasks.

Limitations

Our work has several limitations. First, differential experience effectiveness depends on error pattern coverage; for niche DBMSs with limited samples, the corrective signal is constrained. Second, the harness offers no immediate benefit during cold-start: with an empty store, the system degrades gracefully to the baseline, and value ma-

terializes as memory accumulates. This makes HarnessDB less attractive for one-shot translation than for sustained testing campaigns. Third, the current knowledge base covers three feature categories and four MOLT methods; extending to richer SQL facilities and additional oracle types is an important direction. Finally, the template matching relies on clause-level abstraction, which may be too coarse for highly heterogeneous SQL with deeply nested subqueries or complex CTEs.

References

Jinsheng Ba and Manuel Rigger. 2023a. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Jinsheng Ba and Manuel Rigger. 2023b. Qpg: Automatic generation of query predicates for metamorphic testing of database systems. *arXiv preprint arXiv:2312.17510*.

Tsong Yueh Chen, Shing Chi Cheung, and Siu-Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong*.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and 1 others. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, and 1 others. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Ziyu Cui, Wensheng Dou, Qian Dai, and 1 others. 2022. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. QLoRA: Efficient finetuning of quantized language models. *Advances in Neural Information Processing Systems*, 36.

Zhangyin Feng, Daya Guo, Duyu Tang, and 1 others. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Jingzhou Fu, Jie Liang, Zhiyong Wu, and 1 others. 2024. Sedar: Obtaining high-quality seeds for dbms fuzzing via cross-dbms sql transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12.

Daya Guo, Qihao Zhu, Dejian Yang, and 1 others. 2024. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Zhaoguo Hao, Qiong Huang, Chengpeng Wang, and 1 others. 2023. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 345–358.

Edward J Hu, Yelong Shen, Phillip Wallis, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *ICLR*.

Lei Huang, Weijiang Yu, Weitao Ma, and 1 others. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Ali Reza Ibrahimzada, Yuhang Chen, Ryan Rong, and 1 others. 2023. Automated bug generation in the era of large language models. *arXiv preprint arXiv:2310.02407*.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376.

Yu Jiang, Jie Liang, Fuchen Ma, and 1 others. 2024. When fuzzing meets LLMs: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 492–496.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, and 1 others. 2023. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Marie-Anne Lachaux, Baptiste Rozière, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611.

Caroline Lemieux, Ralf Puddu, Keshav Arya, and 1 others. 2022. General differential testing for database systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Jianyang Gu, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Tianyang Wang, Jihun Hamm, Smita Krishnaswamy, Jun Huan, and Chandan Reddy. 2026. Agent harness engineering: A survey.

Raymond Li, Loubna Ben Allal, Yangtian Zi, and 1 others. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research*.

Yu Liang, Song Liu, and Hong Hu. 2022. Detecting logical bugs of DBMS with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4309–4326.

Li Lin, Zhaoguo Hao, Chengpeng Wang, and 1 others. 2024. Sqless: Dialect-agnostic sql query simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 743–754.

Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. 2025. Qtran: Extending metamorphic-oracle based logical bug detection techniques for multiple-dbms dialect support. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):731–752.

Andrzej Maćkiewicz and Waldemar Ratajczak. 1993. Principal components analysis (pca). *Computers & Geosciences*, 19(3):303–342.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, and 1 others. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, and 1 others. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. *ICLR*.

Qwen Team. 2025. Qwen3.5 technical report. https://qwenlm.github.io/blog/qwen3.5.

Manuel Rigger and Zhendong Su. 2020a. 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*, pages 1140–1152.

Manuel Rigger and Zhendong Su. 2020b. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

Manuel Rigger and Zhendong Su. 2020c. Sqlancer: Detecting logic bugs in SQL database management systems. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference of Systems, Programming, Languages, and Applications: Software for Humanity*.

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

Manuel Rigger and Zhendong Su. 2022. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 128–136.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and 1 others. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized experience replay. In *ICLR*.

Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824.

Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, volume 98, pages 618–622.

Jenson Song, Wensheng Dou, Ziyu Cui, and 1 others. 2023. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2072–2084. IEEE.

Jenson Song and 1 others. 2024. Dqp: Differential query processing for testing database systems. In *Proceedings of the ACM on Software Engineering*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, and 1 others. 2023. Self-consistency improves chain of thought reasoning in language models. *ICLR*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Zhuangda Wang and 1 others. 2025. Coddtest: Automated test generation for database systems via conditional oracle derivation. In *Proceedings of the ACM on Software Engineering*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, and 1 others. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Shunyu Yao, Dian Yu, Jeffrey Zhao, and 1 others. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, and 1 others. 2023b. ReAct: Synergizing reasoning and acting in language models. *ICLR*.

Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, and 1 others. 2023. Large language models are human-level prompt engineers. *ICLR*.

A Knowledge Base Construction

We select data types, functions, and operators as the three categories of mapping features (Lin et al., 2025), since they jointly cover the surface area where dialect-specific divergence most often manifests in cross-DBMS SQL translation. We update the QTRAN crawling scripts to retrieve feature names, descriptions, and examples from the latest official documentation of all eight target DBMSs. Dialect-specific features are identified by parsing each feature-generated query with an ANSI standard parser[2]: queries that fail the parser but succeed in their native DBMS are flagged as dialect-only. Table 4 summarizes the resulting distribution; the average dialect proportion exceeds 10%, confirming that the standard/dialect distinction is non-trivial and critical for cross-dialect translation.

[2] https://github.com/apache/datafusion-sqlparser-rs

Table 4: Statistics of standard and dialect features across DBMSs.

DBMS	Data Types		Functions		Operators	
	Standard	Dialect	Standard	Dialect	Standard	Dialect
ClickHouse	27	3 (10%)	1175	147 (11%)	25	18 (42%)
DuckDB	34	3 (8%)	550	72 (12%)	33	15 (31%)
MariaDB	26	14 (35%)	381	20 (5%)	35	8 (19%)
MonetDB	34	13 (28%)	295	24 (8%)	35	11 (24%)
MySQL	33	5 (13%)	485	27 (5%)	36	6 (14%)
PostgreSQL	58	2 (3%)	539	67 (11%)	91	100 (52%)
SQLite	7	0 (0%)	115	5 (4%)	24	3 (11%)
TiDB	29	1 (3%)	261	13 (5%)	31	6 (16%)
ALL	248	41 (14%)	3801	375 (9%)	310	167 (35%)

B Feature Aggregation and Restoration Algorithm

B.1 Feature Aggregation Algorithm

Algorithm 2 describes the feature aggregation procedure. Given the set of all migration directions M_{dir}, the feature dictionary F mapping each DBMS to its feature list, a batch size B, and a base LLM, the algorithm iterates over each migration direction (S, T). For each source feature f_s, it partitions the target features into batches and issues parallel LLM queries to determine which target features are semantically equivalent to f_s. Matching pairs are merged in a DSU structure. The initial aggregated feature set C_0 consists of all non-singleton DSU clusters.

Algorithm 2 Feature Aggregation Algorithm

Require: Union-Find set U, migration direction set M_{dir}, feature dictionary F, batch size B, base LLM LLM_{base}
Ensure: Initial aggregated feature set C_0

```
Initialize U ← ∅
for each (S, T) ∈ M_dir do
    for each f_s ∈ F[S] do
        Batch_T ← PARTITION(F[T], B)
        for all b ∈ Batch_T in parallel do
            M_match ← LLM_base(Prompt)
            for each f_t ∈ M_match do
                U.MERGE(f_s, f_t)
            end for
        end for
    end for
end for
C_0 ← {c ∈ U | |c| ≥ 2}
return C_0
```

B.2 Feature Restoration Algorithm

Algorithm 3 describes the restoration procedure. Starting from C_0, at each of K iterations, a low-hallucination LLM inspects each cluster c_i and classifies it as invalid (to be discarded), valid (to be retained as-is), or partially invalid (to be split into sub-clusters). The process converges when the LLM can no longer identify refinements, typically within 3–4 iterations. This iterative refinement mirrors the philosophy of DSPy (Khattab et al., 2023) and automated prompt engineering (Zhou et al., 2023), which optimize pipeline components through structured feedback loops. Figure 6 reports the convergence curve, where the aggregation accuracy rises rapidly in rounds 1–3 and stabilizes by round 4.

Algorithm 3 Feature Restoration Algorithm

Require: Maximum iteration number K, low-hallucination LLM LLM_{adv}, initial feature set C_0

Ensure: Final restored feature set C_{final}

Initialize $C \leftarrow C_0$
for $k = 1$ to K **do**
  $C_{\text{new}} \leftarrow \emptyset$
  for all $c_i \in C$ **in parallel do**
    $R \leftarrow \text{LLM}_{\text{adv}}(\text{Prompt}_{\text{fix}})$
    if $R == -1$ **then**
      continue
    else if $R == 0$ **then**
      $C_{\text{new}} \leftarrow C_{\text{new}} \cup \{c_i\}$
    else
      $\{s_1, s_2, \ldots, s_p\} \leftarrow \textsc{Split}(R)$
      $C_{\text{new}} \leftarrow C_{\text{new}} \cup \{s_1, s_2, \ldots, s_p\}$
    end if
  end for
  $C \leftarrow C_{\text{new}}$
end for
return $C_{\text{final}} \leftarrow C$

C Differential Experience Generation Prompts

To construct the differential experience like human, we prompt GPT-5 mini with the following templates. In all cases, the model is instructed to return only raw SQL (or a short analysis) without markdown wrappers or extra commentary. Placeholders such as `{source_db}`, `{target_db}` or `{source_sql}` are dynamically replaced with actual DBMS names and generated SQL text during the data synthesis loop.

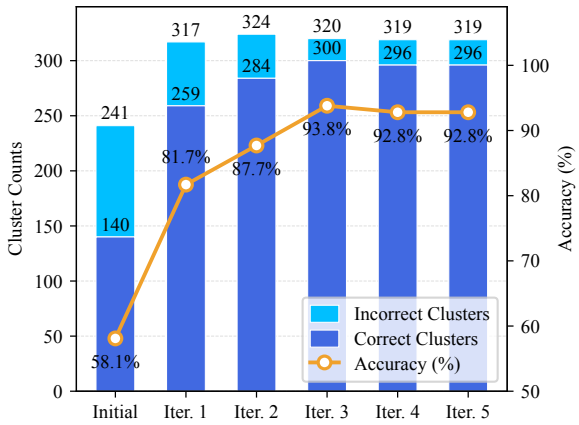

Figure 6: Multi-round feature restoration results. Aggregation accuracy rises rapidly in rounds 1–3 and converges by round 4, achieving a stable final mapping quality.

C.1 Source SQL Generation

System prompt:

```
You are the {source_db} DBMS master.
```

User prompt:

```
[Task Definition]
Generate an minimal EXECUTABLE SQL
    collection around this SQL feature:
{feature_name}, including DDL statements
     and CRUD statements, ensuring
a certain level of SQL richness while
    guaranteeing syntactic and semantic
correctness.

[Return Format]
Return and only return SQL statements
    without any ``` wrapper and
explanation.
```

C.2 Correct Translation

System prompt:

```
You are a SQL translation master,
    proficient in {source_db} and {
    target_db}.
```

User prompt:

```
[Task]
You need to translate {source_db} SQL to
     {target_db} SQL. Here is the
{source_db} SQL: {source_sql}

[Format]
Return complete SQL translation results,
     without any ``` wrapper and
explanation.
```

C.3 Syntactic Error Injection

System prompt:

```
You are a SQL translation master,
    proficient in {source_db} and {
    target_db}.
```

User prompt:

```
[Task Definition]
You need to translate {source_db} to {
    target_db}, but you MUST
    intentionally
introduce the most common, yet subtle,
    human error. Errors MUST ONLY occur
in CRUD statements (SELECT, INSERT,
    UPDATE, DELETE), and strictly NOT in
DDL statements (CREATE, ALTER, DROP).

**DEFINITION OF SYNTACTIC ERROR:**
A syntactic error means the generated
    SQL violates the grammar or syntax
rules of {target_db}, causing it to FAIL
     during parsing or execution. The
error must be extremely subtle and look
    like a typical developer typo.
Examples include (but are not limited to
    ):
- Missing or extra commas.
- Mismatched or missing parentheses.
- Slight typos in SQL keywords or built-
    in function names.
- Incorrect string quoting based on
    dialect.

[Source DBMS SQL]
{source_sql}

[Task Pipeline]
1. Accurately translate the DDL part of
    the given SQL into {target_db}
   without any errors.
2. Translate the CRUD part into the
    target SQL.
3. Inject exactly ONE subtle SYNTACTIC
    error into the CRUD part as defined
   above.

[Return Format]
Complete SQL translation results,
    without any ``` wrapper and
    explanation.
```

C.4 Semantic Error Injection

System prompt:

```
You are a SQL translation master,
    proficient in {source_db} and {
    target_db}.
```

User prompt:

```
[Task Definition]
You need to translate {source_db} to {
    target_db}, but you MUST
    intentionally
introduce the most common, yet subtle,
    human error. Errors MUST ONLY occur
in CRUD statements (SELECT, INSERT,
    UPDATE, DELETE), and strictly NOT in
DDL statements (CREATE, ALTER, DROP).

**DEFINITION OF SEMANTIC ERROR:**
A semantic error means the generated SQL
     is completely valid in syntax and
WILL EXECUTE SUCCESSFULLY in {target_db}
     without crashing, but it contains
a flaw in the business logic, leading to
     INCORRECT query results. The error
must be highly deceptive. Examples
    include (but are not limited to):
- Incorrect join types (e.g., using LEFT
     JOIN instead of INNER JOIN).
- Boundary condition errors (e.g., using
     '>' instead of '>=', or '<'
  instead of '<=').
- Logical operator confusion (e.g.,
    using 'AND' instead of 'OR', or vice
  versa).
- Misplaced aggregations or grouping (e.
    g., applying SUM to the wrong, but
  existing, numeric column).
- Off-by-one errors in date/time math
    functions.

[Source DBMS SQL]
{source_sql}

[Task Pipeline]
1. Accurately translate the DDL part of
    the given SQL into {target_db}
   without any errors.
2. Translate the CRUD part into the
    target SQL.
3. Inject exactly ONE subtle SEMANTIC
    error into the CRUD part as defined
   above, ensuring the SQL remains
       executable.

[Return Format]
Complete SQL translation results,
    without any ``` wrapper and
    explanation.
```

C.5 Semantic Error Analysis

System prompt:

```
You are a SQL Debugging Expert,
    specialized in identifying logical
discrepancies between {source_db} and {
    target_db}.
```

User prompt:

```
[Task Definition]
You are provided with a correct Source
    SQL and a translated Target SQL that
contains an intentional SEMANTIC error.
    Your task is to provide a
professional analysis of this error.

**Context:**
- Source SQL ({source_db}): {source_sql}
- Erroneous Target SQL ({target_db}): {
    sem_err_sql}

**DEFINITION OF SEMANTIC ERROR:**
```

```
A semantic error means the SQL is
    syntactically valid and executes
successfully, but the business logic is
    compromised. It "lies" to the user
by returning incorrect data without
    crashing.

[Task Pipeline]
1. Compare the intended logic of the
    Source SQL with the translated
    Target
   SQL.
2. Identify the specific injection point
     of the semantic error (e.g.,
   changed operator, wrong join type, or
        modified boundary).
3. Explain why this error is "subtle"
    and how it changes the expected
   result set.

[Return Format]
A concise explanation (within 50 words)
    describing the error and its impact
on data accuracy. No ``` wrapper and no
    conversational filler.
```

D Experiment Settings

Table 5 summarizes the detailed experimental configuration. All LLM API calls use OpenAI-compatible endpoints. For translation, we set the temperature to 0 to encourage deterministic outputs. For mutation, the fine-tuned Qwen3.5-9B (Qwen Team, 2025) model is deployed with LoRA (Hu et al., 2022) (rank $r = 8$, $\alpha = 16$) using QLoRA (Dettmers et al., 2024), with the LLaMA-Factory[3] framework and thinking mode enabled to allow step-by-step reasoning about mutation rule application.

Table 6 lists the versions of the eight target DBMSs under test. All DBMS instances are deployed via Docker Compose to ensure reproducible environments.

Table 5: Detailed Experiment Configuration

Phase	Parameter	Value
Global	CPU	Dual Xeon Gold 6530
	OS	Ubuntu 24.04 LTS
	GPU	RTX 5090 (32 GB)
	CUDA	12.8
	CUDA Driver	570.148.08
	Tensor Engine	PyTorch 2.8.0
Data Preparation	RAG LLM	GPT-5 mini
	Temperature	1
	Reasoning Effort	Minimal
Translation	Temperature	0
	Reasoning Effort	Minimal
Mutation	Mutation LLM	fine-tuned Qwen3.5-9B
	Enable Thinking	False
	SFT Framework	LLaMA-Factory
	Learning Rate	5×10^{-5}
	LoRA Rank r	8
	LoRA α	16

Table 6: DBMS Versions Under Test

DBMS	Version
ClickHouse	26.2.4.23
DuckDB	1.5.0
MariaDB	11.8.6
MonetDB	Dec2025
MySQL	8.0.45
PostgreSQL	16.13
SQLite	3.50.4
TiDB	6.5.12

[3] https://github.com/hiyouga/LLaMA-Factory