LLM-based Dynamic Differential Testing for Database Connectors with Reinforcement Learning-Guided Prompt Selection

Ce Lyu[1], Yanhao Wang[1], Jie Liang[2], Minghao Zhao[1,*]
[1]*School of Data Science and Engineering, East China Normal University, Shanghai, China*
[2]*School of Software, Beihang University, Beijing, China*
51275903097@stu.ecnu.edu.cn, yhwang@dase.ecnu.edu.cn, liangjie_work@buaa.edu.cn, mhzhao@dase.ecnu.edu.cn

***Abstract*—Database connectors are critical components that enable applications to interact with database management systems (DBMS) but their security vulnerabilities are often neglected. Unlike traditional software defects, connector vulnerabilities exhibit subtle behavioral patterns and are inherently challenging to detect. Moreover, non-standardized implementation of connectors leaves potential risks (*i.e.,* unsafe implementations) but is more elusive. As a result, existing fuzzing methods are ineffective in finding such vulnerabilities. Even large language model (LLM)-based methods are still incapable of generating test cases that can invoke all the interface and internal logic of database connectors due to a lack of domain knowledge.**

In this paper, we propose a new LLM-based test case generation method guided by reinforcement learning (RL) for database connector testing. Specifically, to equip the LLM with sufficient and appropriate domain knowledge, a parameterized template is composed for prompt construction. The LLM then generates test cases instructed by the constructed prompts, which are dynamically evaluated through differential testing across multiple connectors. The testing process is carried out iteratively, where RL is adopted to select the optimal prompt in each round based on behavioral feedback from the previous rounds, to maximize the efficiency of discovering inconsistencies. Finally, we implement and evaluate the aforementioned methodology on two widely used JDBC connectors, namely MySQL Connector/J and OceanBase Connector/J. In the preliminary results, we have reported 16 bugs, among which 10 are officially confirmed, and the rest are acknowledged as unsafe implementations.

***Index Terms*—differential testing, dynamic testing, database connector, large language models**

I. Introduction

Database connectors (also known as database drivers) serve as intermediaries between applications and databases. They offer standardized interfaces for applications, translating API calls into native database commands while converting query results into application-processable formats. Although this abstraction layer substantially improves development efficiency, its inherent limitations can lead to system-wide failures. Consequently, ensuring the reliability and correctness of the database connectors is critical for overall system robustness.

Nevertheless, detecting vulnerabilities of database connectors is challenging. Unlike traditional software defects, connector vulnerabilities are more elusive and insensitive to traditional software testing technologies. Although state-of-the-art fuzzing techniques have demonstrated remarkable success on DBMS testing [1]–[5], their utility in connector testing remains constrained [6]–[8]. This is because existing fuzzers primarily generate equivalent SQL queries, whereas the connectors forward rather than execute them. Furthermore, the *static* nature of conventional fuzzing renders it ineffective for comprehensively testing connector logic, *i.e.,* as it is difficult for the fuzzer to handle protocol-specific syntax and stateful interactions, the generated queries typically exercise only a limited subset of interfaces and achieve insufficient branch coverage.

What is even worse, certain connector vulnerabilities are scenario-specific or originate from flawed implementation strategies, making such bugs significantly harder to detect. For example, applications often migrate data from one DBMS to another, relying on compatible connectors to handle differences in protocols and SQL dialects. In this process, differences in connector implementations (even for supposedly compatible ones) can trigger exceptions or exhibit inconsistent behavior on additional connectors, such as data loss, silent rollbacks, or duplicate inserts. These subtle deviations may not generate error messages directly and are even more difficult to detect.

Connectors are typically expected to comply with standardized interface specifications, such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). However, in practice, some database vendors deviate from, or even entirely disregard, these standards for certain reasons, *e.g.,* for compatibility with legacy DBMS. Such non-standardized implementations leave potential risks (a.k.a. unsafe implementations) but are more elusive. Furthermore, connector behavior is highly sensitive to property options. These properties can have subtle yet critical effects on query execution, transaction processing, and batch processing. In addition, connector behavior is also dependent on contextual semantics and invocation methods, further increasing the difficulty of writing test cases manually.

Recent research has demonstrated the significant potential and initial success of large language models (LLMs) in software testing [9]–[12]. However, due to the lack of domain-specific knowledge, it is difficult for LLMs to generate test cases that can invoke all the interface and internal logic of database connectors. In addition, static or single prompts often fail to

This work was supported by the National Natural Science Foundation of China (Grant Number 62302256) and Shanghai Sailing Program 23YF1410600.
*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00322

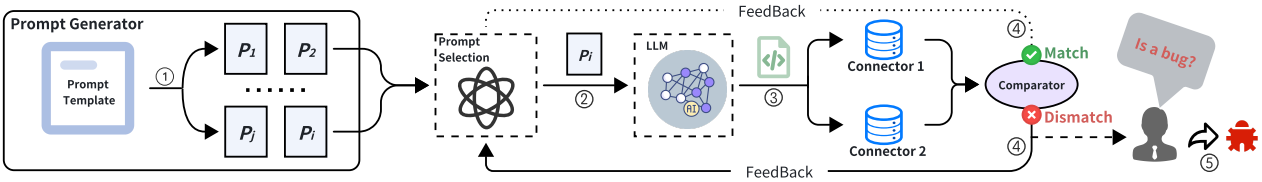

Fig. 1. Overall Workflow of LLM-based Database Connector Testing with RL-Guided Prompt Scheduling.

detect deep vulnerabilities, especially for defects that appear in data transmission between multiple DBMSs.

To address the above limitations, we propose reinforcement learning (RL)-guided LLM-based testing for DBMS connectors, where RL optimizes test-case generation by iteratively refining prompts based on coverage feedback. Specifically, to equip the LLM with sufficient and appropriate domain knowledge, we construct parameterized templates for prompt generation. The LLM then generates test cases based on these prompts, which are dynamically evaluated through differential testing across multiple connectors. The testing process is carried out iteratively, where RL is adopted to select the optimal prompt in each round based on the behavioral feedback from the previous round, to prioritize prompts that are historically effective at finding bugs. By focusing on historically efficient prompts, our approach enables more efficient connector test case generation.

We developed a specialized DBMS testing tool and evaluated its effectiveness by analyzing two widely used JDBC connectors: MySQL Connector/J and OceanBase Connector/J. Our tool detected 16 distinct issues, all of which were reported to the respective maintainers. Among these, 10 issues were officially confirmed as legitimate defects, while the remaining 6 were classified as *unsafe implementations* (as summarized in Table I); these "unsafe implementations" are implementations that do not follow the current JDBC specification [13] due to compatibility with the erroneous behavior of an older version of MySQL Connector/J. Notably, these issues evade detection by established testing tools (*e.g.,* SQLancer), highlighting the unique capability of our approach.

II. Methodology

A. Overview

In this section, we propose a novel framework that leverages an LLM to automatically generate a diverse suite of test cases. The architecture of the proposed framework is illustrated in Fig. 1. Beginning with the *Prompt Generator*, the framework utilizes a structured prompt template to create a set of prompt candidates, which are designed to cover a broad range of connector behaviors (step ①). Next, the optimal prompt is selected from the candidate set using an RL-guided strategy and fed into an LLM to generate a test case (step ②). The generated test case is executed on two compatible connectors for differential testing (step ③). The *Comparator* then examines the results. If discrepancies are observed, a reward signal is propagated back to the *RL Guidance* to reinforce the next iteration of prompt selection (step ④). Finally, inconsistency-triggering test cases are analyzed, simplified, and reported to the respective development teams for validation (step ⑤).

1. Role Definition (Instruction to LLM)
"You are an expert in Java, JDBC, and database connector testing, tasked with generating code to expose subtle behavioral differences and non-standard implementations."

2. Context Specification (Dynamically Provided Inputs)
- *RL-Guided Test Focus*
  - Database Connection URL (including specific properties)
  - Database Schema: Relevant table structures
  - Prioritized Advanced JDBC Operations
  - Targeted Exception/Boundary Scenarios

3. Task Decomposition & Design Directives (Instructions for Test Case Generation)
- *Overarching Generation Philosophy*
  "Generate a "chaotic yet complete" Java test case simulating complex, interwoven, and sometimes unconventional real-world JDBC usage patterns."
- *Code Style & Structural Requirements*
- *Semantic & Behavioral Structure*
- *JDBC API Usage*
- *Test Scenario*

4. Output Format & Execution Requirements (Instructions for final generated case)
- *Code Characteristics*
- *Observability for Differential Testing*
- *Clarity and Conciseness*

Fig. 2. Prompt template for DBMS connector testing.

To facilitate high-quality test generation, our framework employs a structured prompt template that instructs the LLM to simulate the behavior of an expert DBMS tester. As detailed in Fig. 2, the template comprises four key components: role definition, dynamic context specification, task decomposition, and output requirements. This design enables the LLM to generate complex and targeted test cases, thereby detecting bugs and unsafe implementations in the connector. While all prompts share the same structure, we systematically vary the dynamic context and task objectives to instantiate diverse candidates to target different JDBC behaviors and serve as the scheduling units for RL selection.

B. Database Connection Property

The behavior of database connectors is significantly influenced by their connection-level property parameters, such as `allowMultiQueries` and `rewriteBatchedStatements` in JDBC. These options affect the internal optimization paths of connectors, query rewriting logic, and exception handling mechanisms. However, many existing test generators treat the JDBC URL as static, failing to explore the rich behavioral variations induced by different connection property settings. Thus, they miss a critical axis of behavioral variability.

To address this limitation, guided by domain knowledge, we design a systematic connection property module that explores a diverse set of JDBC parameters during test generation. Our goal is to maximize the behavioral surface exposed to the downstream connectors under test. We begin by identifying a set of predefined m JDBC parameters. Formally, we define a property schema $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$, where each c_j corresponds to a Boolean or enumerated JDBC parameter in a well-defined domain. Rather than exhaustively searching the space V of all possible combinations of parameters, *i.e.,*

TABLE I
NUMBER OF BUGS AND UNSAFE IMPLEMENTATIONS.

Type	Database Connector	Quantity
Bugs	MySQL / OceanBase	7 / 3
Unsafe Implementations	OceanBase	6
Total (Bugs + Unsafe Implementations)		16

$\mathcal{V} = \text{Dom}(c_1) \times \text{Dom}(c_2) \times \cdots \times \text{Dom}(c_m)$, we alternatively construct a selected set $\mathbb{S}$ of k representative property vectors, This manual selection was not arbitrary; instead, the set $\mathbb{S}$ includes properties documented to fundamentally alter connector behavior. The set $\mathbb{S}$ is defined as

$$\mathbb{S} = \{v_1, v_2, \ldots, v_k\}, \quad v_i \subseteq \mathcal{V}, \tag{1}$$

where each v_i is a selected connection property vector designed to capture different behaviors. By running tests on a vector of these properties, we can reveal different execution behaviors without exhaustively covering the entire property space. This approach enables a broad yet tractable exploration of connector behaviors under diverse properties, effectively revealing latent issues that are missed because of default settings.

C. RL-Guided Prompt Scheduling

To maximize the effectiveness of LLM-generated JDBC test cases, we introduce an RL-guided prompt scheduling mechanism that adaptively selects prompts based on their historical mismatch-finding performance. We model each prompt template as a distinct arm and apply the Upper Confidence Bound (UCB1) algorithm [14] to guide prompt selection over multiple testing rounds.

We define a set of prompts $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$, where $N \in \mathbb{N}^+$ is the total number of available prompt instances. This set $\mathcal{P}$ is generated systematically from our parameterized template (as illustrated in Fig. 2) to span a range of JDBC features. Our objective is to adaptively identify and prioritize the prompt (P_i) that maximizes the discovery of differential behaviors (indicative of potential bugs) in the database connectors. Each prompt P_i is treated as an arm in the MAB setting, with an unknown reward distribution corresponding to the likelihood that test cases generated from P_i expose connector inconsistencies. We employ the classic UCB1 algorithm to balance exploration and exploitation in prompt selection. In each round, the algorithm selects the prompt P_i that maximizes

$$\mu_i + \sqrt{\frac{2 \log R}{s_i}}, \quad i \in N. \tag{2}$$

For each prompt P_i, we maintain the following variables:

- s_i: The number of times P_i has been selected;
- μ_i: The empirical mean of a binary reward, where the value "1" indicates that the generated test case triggered any observable inconsistency, and "0" otherwise;
- R: The total number of iterations so far.

Once a prompt is selected, it is used to generate a test case via the LLM. The test is automatically rewritten, compiled, and executed against two different JDBC database connectors. A binary reward of 1 is assigned if any behavioral discrepancy is found (0 otherwise), and this reward is used to update s_i and μ_i. This closes the feedback loop between prompt selection and the identification of potentially problematic test scenarios, allowing learning-based scheduling to focus more on prompts with higher impacts.

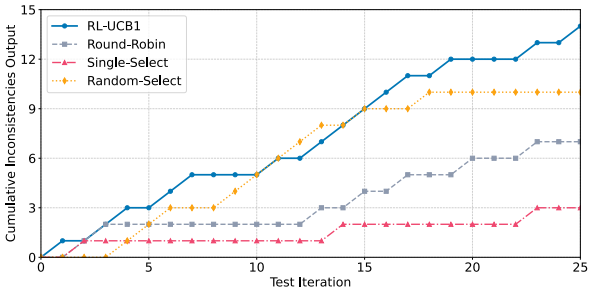

Fig. 3. Comparison of cumulative inconsistencies discovered by different prompt scheduling strategies over 25 iterations.

III. PRELIMINARY EXPERIMENTS

To evaluate the effectiveness of our method in discovering bugs and unsafe implementations on database connectors, we answer the following question: *How does our approach perform on real-world database connectors?*

Tested Database Connectors. We tested two widely used JDBC connectors, namely MySQL Connector/J (v9.2.0) [15] and OceanBase Connector/J (v4.2.0) [16]. We used Qwen-plus [17] as the LLM for database connector test case generation.

Confirmed Bugs & Unsafe Implementations. Table I shows the statistics of our results: seven bugs in MySQL and three bugs, along with six unsafe implementations, in OceanBase. Table II summarizes each issue, including its trigger and description. OceanBase development team explained the reason for these unsafe implementations, stating in an official email *"objdbc does not report errors because it is compatible with the erroneous behavior of MySQL-jdbc 5.x. It will be compatible with 8.x in the future"*. Surprisingly, Issue 3 has remained unfixed in MySQL Connector/J for over 16 years!

Comparison of RL-Guided Prompt Scheduling with Baselines. To evaluate the effectiveness of our RL-guided prompt scheduling, we compared RL-UCB1 with three baseline strategies over 25 iterations with 5 prompts. This evaluation serves as a proof-of-concept rather than a large-scale benchmark. In the experiment, each round corresponds to one full iteration of our workflow (see Fig. 1). The baseline strategies are (1) **Round-Robin**, which cycles through prompts sequentially without adaptation, (2) **Random-Select**, which selects one prompt at random in each round, and (3) **Single-Select**, which randomly selects one prompt and reuses it throughout all rounds.

We used the number of output mismatches between two connectors on the same LLM-generated test case as our evaluation metric. As shown in Fig. 3, RL-UCB1 consistently outperformed the baselines, uncovering 14 inconsistencies in total. Random-Select found 10, Round-Robin 7, and Single-Select only 3. These results demonstrate that RL-based scheduling effectively prioritizes high-yield prompts, thereby improving bug discovery efficiency compared to non-adaptive baselines.

TABLE II
SUMMARY OF BUGS AND UNSAFE IMPLEMENTATIONS IN MYSQL AND OCEANBASE CONNECTOR/J.

ID	Type	Database	Key Aspect / Trigger	Description
Issue 1	🐞	MySQL	Exception Message Error	A message error is output when `setMaxRows()` is used.
Issue 2	🐞	MySQL	Specification Violation	Using `executeBatch()` on a non-DML statement returns an illegal value instead of throwing an exception.
Issue 3	🐞	MySQL	API Behavior	Calling `getHoldability()` is expected to get 1 but actually throws an exception.
Issue 4	🐞	MySQL	Configuration Interaction	The `rewriteBatchedStatements` connection property unexpectedly affects query results following batch inserts.
Issue 5	🐞	OceanBase	Resource Management	`ResultSet` should be closed but is unexpectedly not closed in OceanBase.
Issue 6	🐞	MySQL	Configuration Interaction	The `allowMultiQueries` connection property unexpectedly affects the result of `getUpdateCounts()` after batch execution.
Issue 7	🐞	MySQL	Configuration Breaks Atomicity	Atomicity of batch operation is compromised by `allowMultiQueries`.
Issue 8	🐞	OceanBase	Configuration Interaction	The `rewriteBatchedStatements` connection property unexpectedly affects query results following batch inserts.
Issue 9	🐞	OceanBase	Specification Violation	Using `executeBatch()` to execute a non-DML statement returns an illegal value rather than throwing an exception.
Issue 10	🐞	MySQL	API Behavior	When `resultSetHoldability` has been set to 2, `getResultSetHoldability()` still returns 1.
Issues 11–16	💣	OceanBase	Non-standard Implementation	OceanBase compatibility with erroneous behaviors of MySQL JDBC 5.x: `previous()`, `first()`, `afterLast()`, `absolute()`, `last()`, and `beforeFirst()` do not conform to JDBC documentation [13], which requires throwing `SQLException` when called on a `TYPE_FORWARD_ONLY ResultSet`.

[a] 🐞: Confirmed Bug. [b] 💣: Acknowledged Unsafe Implementation.

Comparison with Existing DBMS Fuzzing Techniques. We adapted SQLancer [2], a state-of-the-art JDBC-based database testing tool, to support multiple database connectors and evaluated both approaches over 100 rounds. The evaluation metric was the number of behavioral inconsistencies observed. While our method detected 33 inconsistencies, SQLancer, despite its extensive fuzzing capabilities, found none. This highlights that existing DBMS fuzzers, which are designed primarily to test DBMS query engines, are ineffective in finding bugs in the connector layer. Thus, our method targets a critical yet often neglected layer of the database software stack.

Case Study. To concisely demonstrate our findings, we present two representative cases: an unsafe implementation in OceanBase Connector/J that deviates from the JDBC specification and a bug in MySQL Connector/J caused by the connection property. We simplified the logic of test cases to highlight the core issues that trigger errors.

Case 1: Listing 1 illustrates a crucial inconsistency. When `beforeFirst()` is invoked by a `ResultSet` instance created with `TYPE_FORWARD_ONLY`, MySQL correctly throws a `SQLException` as required by the JDBC specification [13]. In contrast, OceanBase executes the same call without throwing any exception. This non-standard implementation compromises database portability and may introduce security vulnerabilities.

```
con = DriverManager.getConnection(url);
stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY);
stmt.executeUpdate("CREATE TABLE t0 (Id INT);");
rs = stmt.executeQuery("SELECT Id FROM t0 WHERE Id > 0");
rs.beforeFirst(); 💣
// MySQL triggers SQLException. | OceanBase succeeds.
```

Listing 1. MySQL vs OceanBase: Inconsistent beforeFirst()

Case 2: Listing 2 demonstrates a bug found when inserting duplicate primary keys (1, 1, and 2) into a table $t0$. When the connection property `allowMultiQueries` is enabled, the atomicity of the batch is compromised. Ideally, batch results for primary key conflict should be consistent whether it is enabled or not. The MySQL development team responded: *"Regardless of what the documentation says about the connection property allowMultiQueries, it does affect batched statements"*.

```
con = DriverManager.getConnection(url);
stmt = con.createStatement();
stmt.execute("CREATE TABLE t0 (Id INT PRIMARY KEY);");
stmt.addBatch("INSERT INTO t0 VALUES(1);");
stmt.addBatch("INSERT INTO t0 VALUES(1);");
stmt.addBatch("INSERT INTO t0 VALUES(2);");
stmt.executeBatch(); 🐞
print(); // Assuming prints content of t0
// When allowMultiQueries=true -> print: 1
// When allowMultiQueries=false -> print: 1 2
```

Listing 2. MySQL: Batch Bug with allowMultiQueries Setting

In summary, both cases confirm that our approach can detect bugs and unsafe implementations in database connectors.

IV. CONCLUSION AND FUTURE WORK

As a preliminary exploration, this work addresses a central question, *i.e.,* "*Can our proposed framework detect meaningful database connector vulnerabilities in real-world scenarios?*" Our findings show that LLM-guided differential testing can reveal deep and persistent flaws in connector logic, supporting the feasibility of this approach for real-world systems.

Building on this foundation, our future work will focus on three key areas. First, we will enhance our RL strategy by exploring more expressive reward signals, such as novelty and code coverage, to balance between bug-finding efficiency and broader exploration. Second, we plan to investigate automated techniques for generating high-impact prompts and connection property configurations, reducing the reliance on manual design. Finally, expanding our evaluation to other connectors (e.g., MariaDB and PostgreSQL) will be crucial for validating the generalizability of our framework. We also aim to incorporate coverage-based metrics and assess the cost-efficiency of LLM-based test generation in broader testing pipelines.

REFERENCES

[1] Z.-M. Jiang, J.-J. Bai, and Z. Su, "DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation," in *32nd USENIX Security Symposium*, 2023, pp. 4949–4965.

[2] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 667–682.

[3] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: Testing database management systems with language validity and coverage feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.

[4] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free DBMS fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 49:11–49:12.

[5] W. Deng, J. Liang, Z. Wu, J. Fu, M. Wang, and Y. Jiang, "Coni: Detecting database connector bugs via state-aware test case generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering*, 2025, pp. 26–37.

[6] J. Fu, J. Liang, Z. Wu, Y. Zhao, S. Li, and Y. Jiang, "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*, 2025, pp. 1061–1076.

[7] J. Song, W. Dou, Y. Zheng, Y. Gao, Z. Cui, W. Wang, and J. Wei, "Detecting schema-related logic bugs in relational DBMSs via equivalent database construction," *Proc. VLDB Endow.*, vol. 18, no. 7, pp. 2281–2294, 2025.

[8] Z. Cui, W. Dou, Y. Gao, R. Yang, Y. Zheng, J. Song, Y. Feng, and J. Wei, "Simple testing can expose most critical transaction bugs: Understanding and detecting write-specific serializability violations in database systems," *Proc. VLDB Endow.*, vol. 18, no. 8, 2025.

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 04, pp. 911–936, 2024.

[10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.

[11] F. Qi, Y. Hou, N. Lin, S. Bao, and N. Xu, "A survey of testing techniques based on large language models," in *Proceedings of the 2024 International Conference on Computer and Multimedia Technology*, 2024, pp. 280–284.

[12] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv:2406.00515*, 2024.

[13] Oracle Corporation, "ResultSet.beforeFirst() Method," https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html, 2014.

[14] D. Bouneffouf, "Finite-time analysis of the multi-armed bandit problem with known trend," in *2016 IEEE Congress on Evolutionary Computation*, 2016, pp. 2543–2549.

[15] Oracle Corporation, "MySQL Connector/J developer guide," https://dev.mysql.com/doc/connector-j/en/, 2025.

[16] OceanBase Team, "OceanBase Client for Java," https://github.com/oceanbase/obconnector-j, 2025.

[17] Alibaba Cloud, "Qwen API reference," https://www.alibabacloud.com/help/en/model-studio/use-qwen-by-calling-api, 2025.