On the Flakiness of LLM-Generated Tests for Industrial and Open-Source Database Management Systems

Alexander Berndt*
alexander.berndt@uni-heidelberg.de
Heidelberg University
Germany

Thomas Bach
thomas.bach03@sap.com
SAP
Germany

Rainer Gemulla
rgemulla@uni-mannheim.de
University of Mannheim
Germany

Marcus Kessel
marcus.kessel@uni-mannheim.de
University of Mannheim
Germany

Sebastian Baltes
sebastian.baltes@uni-heidelberg.de
Heidelberg University
Germany

Abstract

Flaky tests are a common problem in software testing. They produce inconsistent results when executed multiple times on the same code, invalidating the assumption that a test failure indicates a software defect. Recent work on LLM-based test generation has identified flakiness as a potential problem with generated tests. However, its prevalence and underlying causes are unclear. We examined the flakiness of LLM-generated tests in the context of four relational database management systems: SAP HANA, DuckDB, MySQL, and SQLite. We amplified test suites with two LLMs, GPT-4o and Mistral-Large-Instruct-2407, to assess the flakiness of the generated test cases. Our results suggest that generated tests have a slightly higher proportion of flaky tests compared to existing tests. Based on a manual inspection, we found that the most common root cause of flakiness was the reliance of a test on a certain order that is not guaranteed ("unordered collection"), which was present in 72 of 115 flaky tests (63%). Furthermore, both LLMs transferred the flakiness from the existing tests to the newly generated tests via the provided prompt context. Our experiments suggest that flakiness transfer is more prevalent in closed-source systems such as SAP HANA than in open-source systems. Our study informs developers on what types of flakiness to expect from LLM-generated tests. It also highlights the importance of providing LLMs with tailored context when employing LLMs for test generation.

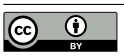
CCS Concepts

• **Software and its engineering → Empirical software validation**; *Software evolution*; *Development frameworks and environments*; **Software reliability**.

Keywords

Test Flakiness, Software Testing, Empirical Study, Database Management Systems, Large Language Models, Artificial Intelligence

*Also with SAP.



ICSE-SEIP '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786919

ACM Reference Format:
Alexander Berndt, Thomas Bach, Rainer Gemulla, Marcus Kessel, and Sebastian Baltes. 2026. On the Flakiness of LLM-Generated Tests for Industrial and Open-Source Database Management Systems. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3786583.3786919

1 Introduction

Large language models (LLMs) have emerged as a promising tool to automate test generation [70]. Although traditional test generation approaches, such as search-based techniques, have been shown to produce high-coverage test suites [34, 58], the resulting test code suffers from poor readability and does not integrate well with existing tests [16, 24, 59]. In contrast, LLMs have the ability to generate test code that appears natural to humans [24, 39, 60]. Hence, various LLM-based test generation approaches have been evaluated [29, 70], leading to industrial adoption of such approaches [5, 37].

To evaluate the resulting tests, existing studies on LLM-based test generation typically measure the effectiveness of LLM-generated tests based on metrics such as code coverage and the mutation score, that is, quantifiable signals regarding the quality of the system under test [22, 70]. However, to our knowledge, there exists no dedicated research that focuses on the *flakiness* of LLM-generated tests. Test flakiness is a major problem in industrial software testing, which has been studied by large software companies such as Google, Meta, Microsoft, and SAP [14, 35, 40, 46]. Flaky tests produce inconsistent results when executed multiple times under the same conditions [50]. Such flaky behavior is undesirable as it reduces the efficiency and effectiveness of testing, thus leading to increased testing costs [12, 35, 53].

Flaky tests invalidate the assumption that a test failure indicates the presence of a defect in the system under test. Thus, developers may lose trust in the test suite and waste time debugging spurious failures [28]. In continuous integration (CI) pipelines, flaky tests impede the automated assessment of code changes. For example, Google reported that the CI pipeline in their testing infrastructure fails in 3.5% of runs due to flaky failures [40].

Previous research on LLM-based test generation indicates that LLMs might produce flaky tests [5, 64]. However, the prevalence of flaky tests and the root causes of flakiness remain unclear. As a practical solution, previous work suggested repeatedly executing

LLM-generated tests to filter flaky tests before handing them to developers for review [6]. Although this strategy can be helpful to filter out flaky tests, we consider it important to obtain additional information and a more holistic view of how and why LLM-generated tests are flaky [14]. Gaining insight into the flaky tests that LLMs produce can help practitioners set realistic expectations about the types of flakiness they may encounter when reviewing and using LLM-generated test code [28, 33]. Furthermore, comparing the prevalence of flakiness in LLM-generated tests with existing tests can help project the long-term flakiness rate of a test suite when LLMs are increasingly used to automatically generate tests.

In this work, we have evaluated the flakiness of LLM-generated tests for relational database management systems. Previous research in the context of the large industrial database system SAP HANA has identified test flakiness as a major issue for automated testing [8, 12, 14, 15]. Motivated by this previous work, we focused on test generation for SAP HANA. In addition, we added three popular open-source databases to foster reproducibility and enable comparisons between open-source and closed-source software [11]. We consider this comparison valuable, as the source code of SAP HANA was not included in the training data, which might impact the results of our experiments.

We adapted Alshawan et al.'s approach for LLM-based test generation, which was previously applied in an industrial study at Meta [5], to amplify tests in the four target systems and generate new test cases. Our study focused on two types of tests: i) native unit tests written in C++ , and ii) SQL system tests. We generated new tests for more than 5000 test files using two LLMs, GPT-4o and Mistral-Large-Instruct-2407 [56, 68]. In accordance with previous empirical studies on flakiness [14, 33, 61, 63], we compiled and repeatedly executed the tests to obtain information on their flakiness. We compared the prevalence and root causes of the resulting flaky tests based on a manual inspection of more than 100 samples.

Our results suggested that LLMs struggle to produce compilable C++ code, especially in the closed-source context of SAP HANA. We attribute this finding to the model's limited understanding of the provided code bases. In LLM-generated tests, the prevalence of flaky tests was often higher than in existing tests. Based on our manual inspection, we identified the reliance of tests on ordered test results from a collection that does not guarantee an order ("unordered collection") as the most common root cause of flakiness in LLM-generated tests for database systems.

To test the robustness of LLMs against flakiness transfer from existing tests, we injected a flaky assertion into the original test code before providing it as input for generating additional tests. In the generated tests, we found that both evaluated LLMs are susceptible to transferring the injected flakiness to generated tests. Although LLMs transferred the injected flaky assertion to two-thirds of the newly generated SAP HANA tests, only half of the generated MySQL tests were affected. Therefore, we conjecture that LLMs rely more heavily on the input context in a closed-source project, for which the source code was not present in the training data. Based on a discussion of these results with practitioners, we conclude that developers should prioritize fixing flaky tests before applying LLM-based generation approaches.

In summary, our study provides the following contributions. We report (1) insights into the prevalence and root causes of flakiness in LLM-generated tests in the context of database management systems, (2) a comparison of the resulting flakiness between closed- and open-source database management systems and between two LLMs, and (3) a benchmark on the susceptibility of LLM to transfer flakiness from the given context in a prompt to the generated tests.

We provide our code and datasets for the three open-source projects in a replication package [13].

2 Background

In this section, we provide the flakiness definition that we adopted and flakiness categories motivated by prior work.

2.1 Flakiness Definition

Flaky tests produce inconsistent results, i.e., yield at least one *positive* and one *negative* result, when executed multiple times under the same conditions on the same state of the source code [33]. More formally, let $R_t = [r_1, ..., r_n]$ be the sequence of observed test executions for a test t on the same state of the source code, where r_i denotes the result of the i-th execution. Furthermore, let $r_i = 0$ for negative results and $r_i = 1$ for positive results. In accordance with previous work on flakiness, we considered a test t flaky iff $0 < \sum_i^n r_i < n$, i.e., if the test did show both positive and negative results [14]. We viewed *passed* as the only positive result, and considered *failed*, *crashed*, or *timed out* negative results. We ignored *skipped* executions. Tests were mainly skipped due to mismatched environmental preconditions, such as a specific operating system.

2.2 Types of Flakiness

Flaky tests can arise due to a variety of reasons. We focused on flakiness from the intra-flakiness family as introduced by Parry et al. [62]. We added *uninitialized variable* and *non-idempotent-outcome* (NIO) Flakiness as we considered them relevant for our experiments. This results in the following set of flakiness categories.

Unordered Collection. Incorrect assumptions on a specific order in an inherently unordered data structure, such as a hash table [62].
NIO. Side effects of a test execution persist in the execution environment, contaminating subsequent test executions [72].
I/O. Improper handling of file system operations, for example, when tests fail due to the disk running out of free space [62].
Uninitialized Variable. Use of variables that have not been properly initialized [32, 52].
Time. Precision issues caused by improper use of system time [62].
Concurrency. Unsafe handling of threads, leading to problems such as race conditions [62].
Randomness. Use of random data generators without proper seeding, leading to inconsistent results [62].
Floating Point. Inaccuracies in floating point operations, such as numerical overflows or underflows [62].
Too Restrictive Range. Too narrow assertions, only allowing a subset of the valid outputs of the system under test [62].
Test Case Timeout. Upper limit on execution time too low, causing intermittent test failures due to execution time variance [62].

2.3 Flakiness of LLM-Generated Tests

In the following, we provide an overview of related work on LLM-based test generation that mentions flakiness as a potential problem.

Alshahwan et al. [5] filtered flaky tests by repeatedly executing LLM-generated tests five times before presenting them to developers for review. In their experiments, 76% of the Kotlin tests that built correctly passed reliably across five executions. However, this proportion also included tests that failed in all five executions, and the exact proportion of flaky tests remains unclear.

Pizzorno and Berger [64] pointed out that LLM-generated tests can suffer from flakiness if the LLM is not provided with sufficient context. They report an example in which GPT-4o generated a flaky test that randomly assigns a value to the property of an object. Since the list of valid values of this property was unknown to the model, this random operation led to intermittent failures when the property was assigned an invalid value. However, since the main focus of their study was LLM-based test generation rather than flaky tests, they provided no information on how often such cases occurred or any other information on test flakiness.

In their framework for benchmarking LLMs for test generation, Azanza et al. [9] introduced *test isolation* as an important quality criterion of the resulting tests. Thereby, they aimed to prevent tests that are flaky due to a high reliance on external dependencies or due to inter-dependencies with other tests. Expert judges in the longitudinal study found that most LLMs are proficient in generating isolated tests, with GPT-4 being the only exception.

3 Database Management Systems

The goal of our study was to gain insights into the flakiness of LLM-generated tests in the context of database management systems (DBMS). The observed types of flakiness may vary for other software projects [32]. We focused our study on two types of tests commonly encountered in database systems: native C++ tests and SQL tests, which execute SQL statements against a running database [10]. We chose SAP HANA and MySQL for our evaluation of native unit tests, as they both share GoogleTest as one of the adopted test frameworks. We further examined DuckDB's and SQLite's SQL tests, as they represent lightweight alternatives offering short build and test times. Table 1 provides a summary of the considered DBMS and included test cases.

3.1 SAP HANA

SAP HANA is an in-memory DBMS that was introduced by SAP in 2010 [10]. Typically, SAP software is used to orchestrate business operations, often using SAP HANA to store and retrieve business-critical data [42]. Thus, failures of SAP HANA can have a negative impact on customers' business operations. This, in turn, justifies high investments in an extensive and continuous testing process.

SAP HANA's source code consists of millions of source lines of code (SLOC), mostly written in C++ [14, 42]. The regression test suite of SAP HANA mainly includes two types of tests: 1) native C++ unit tests that verify a certain functionality in isolation and 2) system tests that execute SQL statements on a running database [10].

Due to the large amount of resources required for testing, SAP utilizes a multistage approach to test SAP HANA [14]. The different testing stages vary in resource demands and the frequency with which the respective tests are executed. These stages range from the local testing of developers in the initial stage to extended manual and automated testing for final release qualification. To cope with the resource demands of the tests executed in the later testing stages, SAP has set up a central testing infrastructure where tests are executed in parallel on over 1000 servers. However, although this central testing infrastructure offers vast computational resources for test execution, previous work has pointed out that such complex testing environments may also cause flakiness, e.g., through hardware failures [14, 51, 66]. In this work, we evaluated the flakiness of DBMS tests, which, according to our definition in Section 2, requires repeated test executions in a controlled environment [33]. To minimize the flakiness caused by environmental factors during repeated test executions [66], instead of relying on the central testing infrastructure, we conducted the test executions on a dedicated server. As executing tests from the later testing stages of SAP HANA would lead to long execution times, we focused on tests that are used for local developer testing, i.e., native C++ tests, written with the GoogleTest framework [31].

3.2 MySQL Server

MySQL Server is an open-source DBMS initially released in 1995 [27]. Today, Oracle offers MySQL Server in two versions: the open-source MySQL Community Server and the proprietary MySQL Enterprise Server. For this work, we examine the open-source MySQL Community Server (in the following referred to as MySQL).

The source code in MySQL's GitHub repository [27] is mostly written in C/C++ and consists of approximately 7 million SLOC, measured using CLOC [3]. Similar to SAP HANA, the MySQL test suite mainly comprises two types of tests: SQL tests written in a dedicated test language and unit tests written in C++. Also similar to SAP HANA, MySQL has a set of unit tests that are written using the GoogleTest framework. To enable a comparison between the results for SAP HANA and MySQL, we examined these GoogleTest unit tests in our study.

3.3 SQLite

SQLite is a lightweight open-source DBMS mostly written in C [20]. According to its documentation, SQLite is the most widely used database engine in the world [19]. The main use case of SQLite is to provide a local data storage for applications that is easy to use and requires no complex setup or administration.

Compared to SAP HANA and MySQL, the SQLite repository contains fewer lines of code, comprising approximately 400 000 SLOC. For testing the SQLite core library, there exist four different test harnesses: TH3, TCL, SQL Logic Test, and dbsqlfuzz [19]. TH3 and dbsqlfuzz are proprietary harnesses and are not publicly available.

Therefore, in our study, we focused on SQLite's Tool Command Language (TCL) test harness, which is publicly available and serves as the main test suite for testing SQLite during development. The TCL test harness provides a TCL wrapper around SQL tests that are executed against a running database instance, similar to SAP HANA's system tests. Thus, our goal was to gain insights into the flakiness of LLM-generated tests that contain SQL statements.

3.4 DuckDB

DuckDB is a lightweight analytical RDBMS developed in C++ [25]. Since its release in 2018, DuckDB has become an emerging DBMS that is continually refined [44, 45, 55]. With its vectorized query

Table 1: The number of relevant test files and test cases in the original test suites of the study subjects. Note that (1) the number of relevant files and tests represents only a subset of SAP HANA's unit tests, which we selected with the assistance of practitioners, and (2) the numbers depend on the different ways in which the test frameworks count tests.

Language	Project	#Test Files	#Tests
C++	MySQL	260	2127
	SAP HANA	1525	19 947
SQL	DuckDB	2748	236 185
	SQLite	603	388 282

execution engine, DuckDB is specifically optimized for online analytical process (OLAP) workloads.

DuckDB's source code is mostly written in C++, comprising almost 2 million SLOC. The DuckDB test suite consists of native C++ tests written with the Catch2 unit testing framework and SQL tests written with sqllogictest [25]. While native C++ tests are specifically targeting the C++ API of DuckDB, DuckDB's developers aim to test most functionality with the help of SQL tests. For our study, we focused on DuckDB's SQL tests to gain insights into the flakiness of LLM-generated SQL tests.

4 Methodology and Research Questions

In this section, we motivate our research questions and outline our methodology for addressing them.

4.1 RQ1: Prevalence

With the first research question, we wanted to gain insights into the prevalence of flakiness in LLM-generated tests.

> **RQ1**: *What is the proportion of flaky tests in LLM-generated tests for test amplification?*

Motivation: Automated test generation (ATG) is one of the most commonly targeted tasks by previous research on LLM-based software testing [70]. Thus, many potential approaches to employ an LLM for test generation have been proposed [4, 5, 18, 22–24, 37, 48, 59, 74], most of them focusing on the generation of regression tests [36]. The goal of this research question was to gain insights into the prevalence of flakiness in LLM-generated regression tests. With this, we aimed to project the long-term impact of an increasing use of LLM-based test generation on the flakiness of a test suite. Existing LLM-based ATG approaches typically utilize LLMs to generate the desired test code directly [4], or in combination with traditional ATG tools such as Pynguin or Evosuite [24, 30, 43, 49]. In this study, we chose to employ a standalone LLM approach, as the scalability of traditional ATG tools to large code bases is limited [41]. Moreover, we did not find any readily available solution for our study subjects, that is large C++ projects.

Approach: We adopted an approach of Alshawan et al. [5], which was previously deployed at Meta with promising results. Figure 1 illustrates our approach, which we chose for three main reasons: i) it was already tested in a large industrial software project with promising results, ii) it amplifies existing test suites and is therefore tailored for projects that already contain a large number of existing tests, and iii) it provides the existing test code as an input for an LLM, which we expected to increase the chance of obtaining compilable and executable output for closed-source projects with highly customized and rather complex test code such as SAP HANA. Furthermore, the chosen approach utilizes a relatively simple setup to generate flaky tests, which enabled its adoption in the context of SAP HANA. In the following, we detail our setup to generate tests for our flakiness evaluation.

Prompt. In accordance with Alshahwan et al. [5], we instructed the model to "Write an extended version of the test file that includes additional tests to cover some extra corner cases.". We provided the existing test file as input context, as previous work has shown that LLMs are capable of generating tests that amplify the current test suite given a minimal context [5]. We chose this rather simple setup, as the focus of our study was only to generate a sample of tests for our flakiness evaluation, assuming that flakiness is evenly distributed between tests generated by different approaches for LLM-based ATG. Finally, we added an output indicator to the prompt so that we could parse the generated code from the LLM's response and patch the existing test file.

Models. We conducted our experiments using two LLMs, GPT-4o and Mistral-Large-Instruct-2407 (in the following: Mistral). Both models were provided by an internal API at SAP, which can be used with an SDK that supports LangChain integration [26], allowing up to 100 requests per minute. We mainly chose GPT-4o because it was the best-performing model [2] available via the internal API, which allowed us to use internal data (i.e., the SAP HANA source code). We repeated our experiments with Mistral's large instruct model to foster the reproducibility of the results for the open-source projects in this study in accordance with existing guidelines on empirical studies using LLMs [11, 69]. We chose Mistral as the open-source alternative as it was the only open-source model available via the internal API at the time of our study. We set the temperature to zero for both models, as this setting has been shown to produce tests with the highest compilation success rate (CSR) in a previous study [5]. Both models have a context window of 128k tokens. While GPT-4o's number of parameters is estimated at 200b [1], Mistral-Large-Instruct-2407 has 123b parameters.

Based on the given prompt and models, we generated code for each of the files listed in Table 1 using both LLMs. After parsing the newly generated test code from the LLMs' response, we extended the original test files with the new code. By integrating the generated code into existing test files, we were able to utilize the test frameworks of the respective projects as-is, which simplified the automated execution of the resulting test code. After patching the original test files, we compiled the resulting code to filter syntactically incorrect files.

Execution Environment. To ensure a clean environment for every test execution and avoid flakiness caused by environmental factors, we executed the tests in separate Docker containers, which is common practice in empirical studies on test flakiness [33, 47, 63, 65, 66, 71, 75]. For this, we created a dedicated Docker image for each of the projects in our study. These Docker images contained a compiled version of the respective project, so after launching a container based on the image, we only had to patch the relevant test file before initiating the test execution. We report the commit

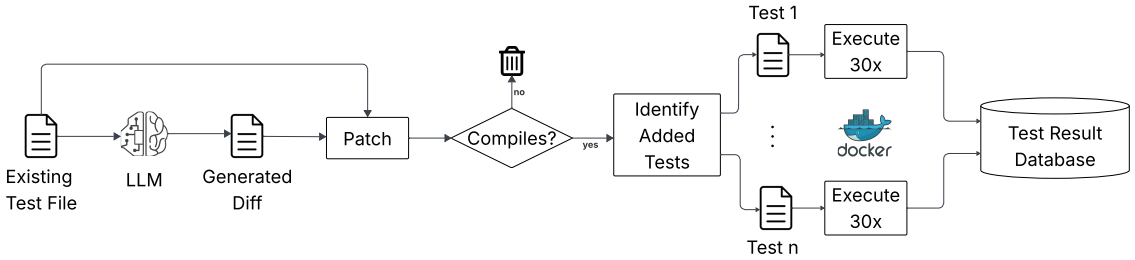


Figure 1: Study setup inspired by Alshahwan et al. [5]: We first prompt the LLM to generate new tests given an existing test file. Then, the generated code is added to the existing test file before compilation. After compilation, we identify the newly added, generated tests, execute them in separate Docker containers, and store the results in a database.

hashes of the projects we used for this study in our replication package. In the following, we briefly describe the execution setup for the two types of tests in this study.

Executing C++ Tests. For the C++ tests, we launched a Docker container for each *test* before executing the tests one-by-one in isolation. In preliminary experiments, we found that "naively" executing generated tests for database systems at scale may lead to crashes of the execution server. For example, a generated test for SAP HANA's memory management tried to allocate a very large amount of memory, thus causing resource contention. We limited the impact of such cases by providing the Docker containers with a fixed amount of computational resources and timing out long-running test executions after five minutes. Note that this specific execution environment also sets limits on the types of flakiness that we could possibly observe. For example, by providing the test executions with a fixed amount of computational resources, we expected to reduce the impact of resource-affected flaky tests [66]. More specifically, we expected mainly observe intra-test flakiness in our experiments, as our test isolation via Docker minimizes the probability of failures caused by inter-dependencies between tests [62]. We made this decision to increase the control and focus of our study and to identify only the types of flakiness that were caused by the generated code itself.

Executing SQL Tests. In contrast to the C++ tests, we launched the Docker containers for the SQL tests for each *test file*. This is due to the structure of the SQL tests. The SQL tests are order-dependent by design, i.e., the custom test frameworks of SQLite and DuckDB always execute the tests in a file in the same order, thus explicitly allowing developers to write order-dependent tests. Listing 1 illustrates an example from DuckDB. In this case, the test aims to test the `join` operator. For this, a table is first created and populated, before the `join` operator is applied. Although one might consider such tests inherently flaky, as executing them in a different order would cause test failures, we emphasize that the focus of our study was on practically relevant flakiness. Therefore, we executed the tests in an environment that resembles the testing environment in practice, where the execution order is predetermined.

Existing Tests. To ensure that our execution environment meets the requirements for the given projects and to identify any potentially existing flaky tests, we first executed all relevant existing tests from the different projects $n = 30$ times and stored the results from these test executions in a database for later analysis. We chose to perform 30 repeated executions, as our goal was to detect flaky tests that are practically relevant, i.e., tests that fail with a high probability.

4.2 RQ2: Root Causes

With the second research question, our aim was to find the most prominent root causes of test flakiness in LLM-generated tests.

> **RQ2**: *What are the flakiness root causes of LLM-generated tests for test amplification?*

Motivation: One might expect that the root causes for the flakiness of LLM-generated tests follow the distribution of existing tests, as LLMs essentially mimic human behavior they learned from their pre-training. However, as discussed in Section 2, previous research suggests that flakiness in LLM-generated tests can occur due to different root causes [9, 64]. For example, LLMs may generate flaky tests due to a lack of sufficient context, which is available to developers but has not been externalized in the context of a prompt. Furthermore, previous work also pointed out that the root causes of flakiness vary between different types of software [32]. Therefore, we investigated the root causes of flaky tests generated by LLMs in the context of DBMS.

Approach: To gain insights into the root causes of flakiness in LLM-generated tests, we manually labeled the flaky tests identified based on our definition of flakiness in Section 2, identified by repeated test executions. We conducted the labeling using the test results of the repeated test executions, the resulting error messages in failing executions if reproducible, and the executed test code. In accordance with previous research on flakiness [33], we utilized the categories by Parry et al. [62] as described in Section 2 for our labeling. To enable a comparison between the root causes of flakiness for tests written by developers and those generated by the LLM, labeling was conducted for the flaky tests resulting from the generation experiments, as well as for existing flaky tests.

4.3 RQ3: Transfer

Our third research question concerned the degree to which LLMs are susceptible to transferring flakiness from existing tests to the newly generated tests.

> **RQ3**: *To what degree does an LLM transfer flakiness present in the provided in-context example tests to the newly generated tests?*

Motivation: Previous work on LLM-based test generation found that when generating new tests, LLMs follow existing coding conventions and adhere to the structure of the code presented in the context of a prompt [5, 37]. Harman et al. [36] called this effect "Fashion Following" and argued that the resulting test code is thus more intuitive to human readers. This "Fashion Following" can be attributed to the ability of LLM to learn from examples in the given input context, that is, in-context learning (ICL) [17, 73]. ICL has been shown to be an effective approach to improve the performance of an LLM on various tasks [17]. However, previous research has also shown that LLMs lack robustness against spurious correlations when learning from the provided context [38, 67]. We hypothesized that, for test amplification, this may lead to the transfer of flakiness from the existing tests, which are shown to the LLM, to the newly generated test code. Therefore, we aimed to gain insights into the robustness of LLMs against such a flakiness transfer.

Approach: We injected flakiness into existing tests and replicated our test generation experiment with the injected flaky tests. More specifically, we added an assertion to the C++ tests that fails with 50% probability, using the following steps [21]: (1) Parse the original code of the given tests from the C++ files. (2) Add an assertion to the test, which may fail based on the outcome of a coin flip at the end of the test code. (3) Replace the original test code with the injected code for every test in all relevant test files. (4) Persist the injected files for later use. We performed these steps before providing the test files to the LLM and evaluated whether the LLM "follows" the injected flakiness in its generated output.

4.4 Threats to Validity

Internal Validity. In contrast to Alshahwan et al. [5], we did not calculate the code coverage of the resulting tests. Instead, we kept all compiling tests for our flakiness evaluation. To mitigate the threat of analyzing mostly nonsensical tests, we conducted a qualitative analysis of the flaky tests during our manual inspection of flakiness categories. Furthermore, we analyzed the syntactic similarity of the generated tests with the existing tests to ensure that the LLM did not simply reproduce existing tests with new inputs at scale. We measured the syntactic similarity with the Levenshtein Edit Distance (LED), as it is a simple but effective measure to compare the syntactical similarity of generated test code [57]. To achieve this, we compared each existing test with each newly generated test in a file. For example, in a file with $n = 3$ existing tests and $m = 2$ LLM-generated tests, we compared both LLM-generated tests with the 3 existing tests, thus ending up with $n \times m = 6$ comparisons. We report the distribution of the closest distances between existing and LLM-generated tests per file in Figure 2.

Since LLMs are inherently non-deterministic, multiple repetitions of the same experiment may lead to different results. To mitigate the threat of the inherent non-determinism of LLM, we followed two approaches. First, we repeated relevant parts of our experiments to measure the degree to which our conclusions could be impacted by randomness. We performed these repetitions with GPT-4o, as it consistently yielded better results than Mistral. More specifically, we repeated the generation for files containing flaky tests in our initial experiments to see whether the flakiness persists,

```
statement ok
CREATE TABLE test (a INTEGER, b INTEGER);
statement ok
INSERT INTO test VALUES (11, 1), (12, 2), (13, 3);
statement ok
CREATE TABLE test2 (b INTEGER, c INTEGER);
statement ok
INSERT INTO test2 VALUES (1, 10), (1, 20), (2, 30);
# simple cross product + join condition
query III
SELECT a, test.b, c FROM test, test2 WHERE test.b = test2.b ORDER BY c;
----
11      1       10
11      1       20
12      2       30
```

Listing 1: Example structure of an SQL test for DuckDB.

Table 2: An overview of the relevant existing tests in our study subjects.

Metric	SAP HANA	MySQL	SQLite	DuckDB
#Tests	19 947	2127	388 282	236 185
#Flaky Tests	13 (0.07%)	13 (0.61%)	0	0

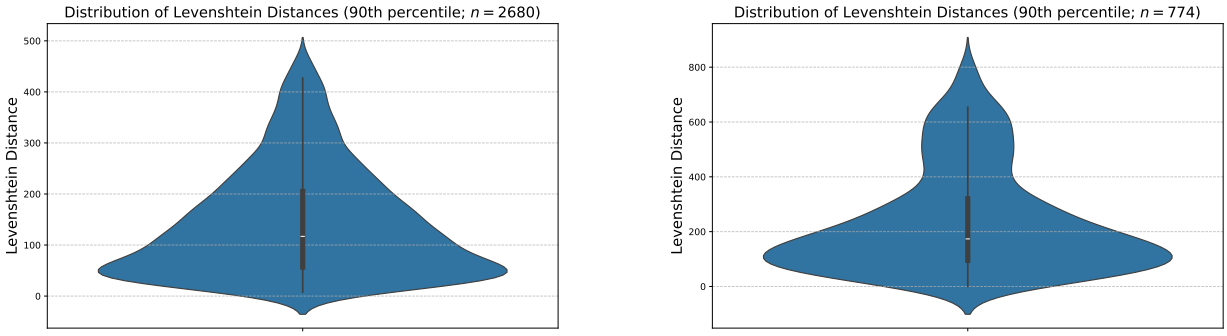
and repeated the injection experiments two additional times to ensure the reported proportion of tests containing the flaky assertion is valid. Furthermore, in all our experiments, we set the temperature to zero, as this setting has been shown to promise the most consistent results in a previous study [57].

Construct Validity. In contrast to previous studies on flakiness, which have performed up to 10 000 repeated executions of a test to gain insights into its flakiness [7], we executed the tests in this study only 30 times. Since some flaky tests have been shown to fail rarely [14], we may therefore underestimate the prevalence of flakiness in our study. For example, assuming that test executions are independent and identically distributed, we expected to identify tests with a failure probability of 15% with a probability of 99%. With 10 000 repeated executions, tests with a failure probability of 0.04% could be identified with 99% probability. However, using 30 repeated executions to identify flaky tests is similar to the reality of SAP's approach for detecting flaky tests, which is based on the trade-off between detecting all flaky tests and overestimating the flakiness due to environmental noise.

External Validity. Our study setup is limited to a single LLM-based ATG approach in the idiosyncratic context of relational database systems. Therefore, the resulting flakiness distributions we observe might not generalize to other contexts, such as NoSQL database systems or different software domains. Furthermore, given the small number of flaky tests in our study, the statistical significance of our results is limited. We view the results of our study as a first step toward more detailed insights into the flakiness of LLM-generated tests. We motivate other researchers to experiment with different prompts, LLM-based ATG setups, and software projects.

5 Results

In this section, we report the results of our experiments. We start by giving an overview of the execution results of existing tests. Afterwards, we report the results of the LLM-generated tests.



(a) Tests generated by GPT-4o using existing tests in SAP HANA. **(b) Tests generated by GPT-4o using existing tests in MySQL.**

Figure 2: Violin plots displaying the syntactical difference between generated and existing tests.

Table 3: The results from the executions of our LLM-generated tests. Note that SQLite and DuckDB tests are written in SQL and do not need to be compiled (compilation success rate marked with "—" in the table).

	Project	#Diffs	#Compilation Successes	#Added Tests	#Passing Tests	#Flaky Tests
GPT-4o	SAP HANA	1525	707 (46%)	2975	1678 (56%)	5 (0.29%)
	MySQL	260	173 (67%)	868	550 (63%)	1 (0.18%)
	SQLite	598	—	133 592	126 774 (95%)	4 (0.00%)
	DuckDB	2688	—	71 031	57 591 (81%)	42 (0.07%)
Mistral	SAP HANA	1525	616 (40%)	3257	1773 (54%)	6 (0.39%)
	MySQL	241	138 (57%)	861	560 (65%)	4 (0.71%)
	SQLite	592	—	60 123	52 276 (87%)	6 (0.01%)
	DuckDB	2676	—	119 666	100 794 (84%)	47 (0.05%)

5.1 RQ1: Proportion of Flaky Tests

In our first research question, our aim was to gain insights into the prevalence of flakiness in LLM-generated tests. In the following, we report the quantitative results of our test executions.

Existing Tests. Table 2 shows the results of 30 repeated executions of the existing tests. We observed that both C++ projects show a small number of flaky tests. For MySQL, 13 of 2127 (0.6%) passing tests show flaky behavior. This number is similar to the flakiness numbers reported in previous work [33]. Analyzing the 13 flaky tests, we observed that they originated from four different test files. They all suffered from NIO flakiness, i.e., the tests self-pollute the execution environment in their first execution. Listing 2 shows an example of an NIO-flaky MySQL test. In the example in Listing 2, the test uses static variables that are not reinitialized in subsequent executions, thus causing the assertions in lines 18 and 19 to fail.

We observed no flaky tests in the existing tests of SQLite and DuckDB. While all original tests for DuckDB yielded a passing result, ten tests of SQLite did not pass in our execution environment. We found that one of these tests exceeded our five-minute timeout, one was skipped, and eight tests failed. All failing tests resulted from a single file containing tests for SQLite's write-ahead logging (WAL) mode [19]. Each of these tests required ten threads simultaneously, which collides with our approach to parallelized test execution. We omitted this file in the following experiments.

Generated C++ Tests. Table 3 shows the execution results of the LLM-generated tests. Using GPT-4o, we generated 1785 diffs resulting in 3843 added tests. For Mistral, we obtained a similar number with 4012 added tests. The compilation success rate (CSR) differed between MySQL and SAP HANA both for Mistral and GPT-4o. While GPT-4o produced compilable code for two out of three files for MySQL, less than half of the resulting files for SAP HANA were compilable. We consider this result intuitive, as SAP HANA's source code was not part of the training data and previous research suggests that missing background knowledge on the project context may cause compilation errors due to hallucination [76]. Comparing Mistral to GPT-4o, we observed a slightly lower CSR for Mistral, with a drop by a factor of 0.87 for SAP HANA and 0.82 for MySQL. Similarly, comparing the pass rates for SAP HANA and MySQL, we observed that tests generated for MySQL yield a slightly higher pass rate (63% vs. 56%) for both LLMs.

We observed a relatively low number of flaky tests for both SAP HANA and MySQL. GPT-4o generated a single flaky test for MySQL. For SAP HANA, we observed flaky behavior in five generated tests. With six flaky tests for SAP HANA and four for MySQL, Mistral generated a higher number of flaky tests, resulting in a slightly higher proportion of flaky tests in both projects.

Generated SQL Tests. We report the results of the SQL tests in Table 3. Compared to the two C++ projects, we observed higher pass

```
  TEST(
    ut0new_new_delete_arr,
    destructors_of_successfully_instantiated_
    trivially_constructible_elements_are_invoked
    _when_one_of_the_element_constructors_throws) {
  static int n_constructors = 0;
  static int n_destructors = 0;

  bool exception_thrown_and_caught = false;
  try {
    auto ptr = ut::new_arr_withkey<Type_that_may_throw>(
        ut::make_psi_memory_key(pfs_key), ut::Count(7));
    ASSERT_FALSE(ptr);
  } catch (std::runtime_error &) {
    exception_thrown_and_caught = true;
  }
  EXPECT_TRUE(exception_thrown_and_caught);
  EXPECT_EQ(n_constructors, 4);
  EXPECT_EQ(n_destructors, 3);
}
```

Listing 2: Example for existing flaky test in MySQL.

rates for SQL tests, ranging between 81% and 95%. Note that there was a large difference in the number of generated tests for DuckDB and SQLite between models. We attribute this difference to the way we count SQL tests. A single SQL test in the code may be executed for different configurations of the database on which the test is executed. Since we counted each instance of such an execution as a distinct test, one additional generated test may increase the number we count by more than one. Although there were no existing flaky tests in the original test suites of both projects, both models generated flaky tests for SQLite and DuckDB. GPT-4o generated a relatively low number of flaky tests for SQLite (4), but a notably higher number of flaky tests for DuckDB (42). With six flaky tests for SQLite and 53 for DuckDB, Mistral generated more flaky tests than GPT-4o, similar to our results for the C++ tests.

Syntactical Similarity. Figure 2 displays the LEDs between existing and LLM-generated tests. We observed an average LED of 299.14 between LLM-generated and existing tests for SAP HANA (min=8, Q_1=62, Q_3=250, max=45 926). Furthermore, for SAP HANA, there was no LLM-generated test that was an exact copy of an existing test. For MySQL, the syntactic similarity between LLM-generated and the closest existing test per test file is higher, with an average of 405 (min=0, Q_1=101, Q_3=466, max=10 154). Overall, we conclude that the resulting LEDs suggest a sufficiently large syntactical dissimilarity between existing tests and tests generated by the LLMs.

Result RQ1 (Proportion): The proportion of flaky tests generated by the LLM varied between projects and models. Compared against existing tests, we observed a higher proportion in the generated C++ tests by GPT-4o for SAP HANA. For MySQL, the proportion of flaky tests was lower for the generated tests. For Mistral, the proportion of flaky C++ tests is slightly higher. Although we found no flaky SQL test written by a developer, we found 96 flaky tests in the tests generated by the LLMs in this study (43 for GPT-4o, 53 for Mistral).

5.2 RQ2: Root Causes

We manually inspected the resulting flaky tests to gain insights into the root causes of their flakiness. We report the results of our manual categorization in Table 4.

```
-- Test with UNION BY NAME and complex data types
query I
SELECT {'key': 'val'} AS json_col UNION BY NAME SELECT {'key': 123} AS json_col;
----
{'key': val}
{'key': 123}
```

Listing 3: Example for a flaky test of the unordered collection category generated by GPT-4o for DuckDB.

```
-- Test with a large number of random values
statement ok
CREATE TABLE random_values AS
SELECT random() * 1000 - 500 AS x FROM range(10000);
query I
SELECT mad(x) FROM random_values
----
250
```

Listing 4: Example for a flaky test of the random category generated by GPT-4o for DuckDB.

Overall, the most prevalent category for flaky LLM-generated tests was "unordered collection", appearing in 33 of 52 flaky tests for GPT-4o (65%) and 38 of 63 tests for Mistral (60%). All flaky tests in the "unordered collection" category were generated for DuckDB. In these cases, the LLMs generated SQL statements that assumed an order for the result of a query, without proper use of `ORDER BY`. Listing 3 illustrates an example of such a test, which was generated for a file containing tests for the `UNION BY NAME` operator. Generally, DuckDB offers two base operators for performing unions, `UNION` and `UNION ALL` [54]. The difference between `UNION` and `UNION ALL` is that `UNION` performs deduplication when stacking, thereby neglecting the original row order. However, in the case of the example in Listing 3, the LLM utilized the `UNION BY NAME` operator instead of `UNION ALL BY NAME`, without an explicit `ORDER BY`. Thus, the test suffered from flaky failures as the values might be returned in the wrong order.

Generally, the distribution of flakiness categories appeared similar between the two models. As shown in Figure 3, there was also a notable overlap in the files that include the generated flaky tests, suggesting that some files are particularly prone to flakiness. Listing 4 shows an example of a test that suffered from randomness, which was generated by both the GPT-4o and the Mistral model for the file `test_mad.test`. In addition to "unordered collection", we found "NIO" flakiness to be a problem for both LLMs, appearing in 12 of 115 flaky tests overall (10%). Nine of these 12 flaky tests were generated for one file in SAP HANA's repository, five by Mistral and four by GPT-4o. In this case, the LLMs employed an existing test function that increases a counter variable without properly cleaning the state at the end of the test. Since this function also causes flakiness in existing tests, the models merely followed the flakiness that was already present in the context.

For tests of the "concurrency" category, there is a notable difference between the models. Eight out of nine tests that suffered from concurrency were generated by Mistral. These tests were generated for SQLite, MySQL, and DuckDB. For SQLite and DuckDB, they tested the respective thread handling implemented in the project. In DuckDB, the issue was related to the locking functionality. In these

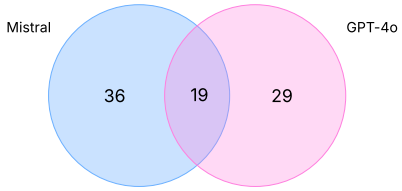


Figure 3: Venn diagram showing the number of files containing generated flaky tests for GPT-4o and Mistral, respectively.

Table 4: Root cause category frequency of flaky tests.

Type	GPT-4o	Mistral	Total
Unordered Collection	34	38	72
NIO	5	7	12
Concurrency	1	8	9
Unknown	4	2	6
I/O	2	3	5
Randomness	1	3	4
Other	3	0	3
Uninitialized Variable	1	1	2
Time	1	1	2
Floating Point	0	0	0
Too Restrictive Range	0	0	0
Test Case Timeout	0	0	0
Total	52	63	115

cases, similar to the tests in "unordered collection", Mistral lacked knowledge of parallelism handling in the respective projects.

Repeated Executions. To gain insights into whether the flaky tests persist in repeated executions of our experiment, we repeated our generation for the files containing flaky tests in the context of SAP HANA and MySQL two additional times. We found that every flaky test in the initial iteration was reproduced in at least one of the repetitions for the C++ tests. For MySQL, the single flaky test observed in our initial experiment was reproduced in both repetitions. For SAP HANA, we found that one flaky test was only reproduced in a single repetition, whereas the semantic copies of the other four flaky tests were contained in both iterations.

For the SQL tests, we observed that three of four flaky tests for SQLite were reproduced in two additional repetitions. For DuckDB, of the 42 flaky tests in the initial experiment, we were able to reproduce 37 in the first and 40 in the second repetition. Overall, we conclude that there was a sufficiently large overlap of the root causes between repetitions.

Result RQ2 (Root causes): Unordered collection was the most prevalent root cause of flaky tests in our study. All instances of tests suffering from this cause were SQL tests, where the LLMs expected an ordered result set without proper use of order by. We also found NIO to be a common root cause of flakiness in LLM-generated tests, mostly caused by LLM following the flakiness of existing tests.

5.3 RQ3: Flakiness Transfer

We report the results of our flakiness transfer experiment in Table 5. As shown in Table 5, both models were susceptible to transferring the injected assertion to the newly generated tests. In addition, GPT-4o and Mistral showed an increased susceptibility to transferring the flaky assertion in the context of SAP HANA compared to MySQL. The proportion of tests containing the flaky assertion was approximately 20% higher between projects for both models. We conjecture that this is due to the background knowledge the models have gained on MySQL during pre-training.

Comparing between models, GPT-4o yielded a lower proportion of tests containing the flaky assertion by a factor of 0.88 for SAP HANA and 0.72 for MySQL. Looking at the difference between passing and non-passing tests, there was only a negligible difference between the proportion of tests containing the flaky assertion for GPT-4o. For tests generated by Mistral, the proportion was lower by a factor of 0.92 for SAP HANA and 0.78 for MySQL. On a file-level, we observed that the LLM most often either transfers the flaky assertion to all newly generated tests in a file, or to none of them. For example, the 1052 resulting flaky tests for SAP HANA generated by GPT-4o arose from 344 generated diffs, i.e., 23% of all generated diffs. However, only one of these diffs contained both tests with and without the flaky assertion. Similarly, the flaky tests generated for SAP HANA by Mistral are contained in two files.

Repeated Executions. Based on two additional repetitions of our experiments, we observed only marginal changes compared to the results in Table 5, with the largest deviation being 2%. For MySQL, the proportion of tests that contained the flaky assertion in our repetitions was 40% and 42% compared to 42% in our initial experiment. For SAP HANA, we observed 68% and 71% compared to 69% in our initial experiment.

Result RQ3 (Following): Both LLMs were susceptible to transferring the flakiness present in the provided context. Compared to GPT-4o, a higher proportion of tests generated by Mistral contained the flaky assertion. For both models, the prevalence was higher in the context of SAP HANA compared to MySQL.

6 Discussion

In this section, we discuss the implications of our empirical results.

On LLM-based Test Generation. Overall, we observed that the LLM-based test generation approach of Alshahwan et al. [5] resulted in a reasonable number of generated tests. Although we chose the simplest possible setup to generate tests using an LLM, we were able to generate compilable diffs, even for complex projects such as SAP HANA. However, compared to the original study of Alshahwan et al. [5], we observed a notable decrease in compilation success. This decrease is expected due to the inherent complexity of database systems and the complexity of C++ compared to Kotlin. The drop in compilation success for SAP HANA confirms our assumption that LLMs may yield worse results in a large and complex closed-source project. Based on the results of our study, future work at SAP is currently exploring approaches to provide LLMs with the required context to handle the large number of custom libraries used within SAP HANA's source code, which are not known to public LLMs. We view the simple setup of this study as a starting point for future work

Table 5: The results from our test amplification approach using the test files with injected flaky assertions.

	Project	#Diffs	#Compilation Successes	#Added Tests	#Tests w/ Flaky Assertion	#Passing Tests	#Flaky Tests
GPT-4o	SAP HANA	1505	638 (42%)	2794	1939 (69%)	1510	1052 (70%)
	MySQL	259	176 (68%)	792	331 (42%)	526	217 (41%)
Mistral	SAP HANA	1505	652 (43%)	2591	2012 (78%)	1467	1051 (72%)
	MySQL	252	142 (56%)	867	507 (58%)	715	319 (45%)

on this topic, which may employ more sophisticated approaches to test generation. For example, future work could experiment with other setups than test amplification or investigate the impact of different prompting techniques.

On the Resulting Proportion of Flaky Tests. We found that both LLMs generate flaky tests across all projects. The resulting proportion of flaky tests for LLM-generated C++ tests was slightly higher than the proportion of flaky tests in the existing tests. Consequently, we expect that the use of LLMs for the test generation could increase the overall proportion of flaky tests in a test suite. Furthermore, fixing flaky tests is often neglected due to other priorities [14, 40], even for existing tests. For generated tests, the lack of ownership for these tests may exacerbate this effect, leaving LLM-generated flaky tests as a long-term liability in the test suite.

On the Results of our Manual Inspection. Based on our manual inspection of flaky tests, we conjecture that a lack of context on the system under test may be the major cause of flakiness in LLM-generated tests. This seems particularly pronounced for SQL tests, where the LLM had no information on when the system under test guarantees the order of query results. While adding related context may be a valid solution in the context of unit tests, where the test's scope is by definition limited, such approaches are challenging to apply to SQL tests, as they may require a large amount of context. For example, with millions of SLOC, SAP HANA's source code does not fit entirely into the context windows of contemporary LLMs. Therefore, we encourage future work on approaches that automatically provide LLMs with tailored context for generating system tests, which requires information about the entire software.

On the Impact of Flakiness Injection. Overall, we observe that the LLM transfers the injected flaky assertion to a notable number of newly generated tests for both study subjects. Since the flaky assertion in the provided context is clearly not causally related to the task at hand, we attribute this finding to the LLM's susceptibility to spurious correlations in in-context learning. We conjecture that the LLM overvalues the provided context compared to the knowledge gained from pre-training. While this may lead to desirable effects such as the "Fashion Following" of existing coding conventions [36], the results of our simple setup suggest that this happens without any semantic understanding of the given code, which may also lead to undesired properties in code synthesized by LLM in practice. Therefore, we encourage future work to investigate this effect. For example, future studies could further validate the degree to which LLMs propagate non-functional code properties, such as code smells, from the code shown for in-context learning.

7 Conclusions

To gain insights into the flakiness of LLM-generated tests for database systems, we used two LLMs to generate two types of tests commonly encountered in database systems: native C++ unit tests and SQL tests that run SQL statements. We applied an approach for LLM-based test amplification to three open-source databases and SAP HANA, a large industrial database, and examined the flakiness of the resulting tests.

Our study revealed that LLMs struggle to generate compilable C++ source code, especially in complex closed-source projects such as SAP HANA. Our analysis showed that the most common root cause of flakiness generated by LLMs in the context of our study was SQL statements in which the LLM expects a certain order without proper use of ORDER BY. We found that the proportion of flaky tests as well as root causes varied between LLMs. However, both LLMs transferred existing flakiness from existing tests in the given context of a prompt to newly generated tests. We attribute this problem to the shortcut learning of LLMs, where they follow spurious correlations in the provided context during in-context learning. To further understand this phenomenon, we injected a flaky assertion into all existing tests before repeating our generation experiment with the injected files. Our results suggest that LLMs are susceptible to transferring the injected assertion. We found that both LLMs were more susceptible to our injection in the context of SAP HANA compared to MySQL. We attribute this finding to the fact that public LLMs lack prior knowledge of SAP HANA, as its source code was not part of their training data, leading to greater reliance on the given context in the prompt.

Our findings substantiate the importance of providing LLMs with appropriate context when employing them for test generation. We motivate practitioners to pay attention not only to engineering goals such as coverage or mutation scores when evaluating LLMs for test generation, but also to consider potential side effects of the resulting tests, such as flakiness. We encourage future studies to evaluate and extend our test generation setup by including additional test generation approaches and models. We also encourage researchers to investigate the impact of shortcut learning on the application of LLMs for automating software engineering tasks.

References

[1] Asma Ben Abacha, Wen-wai Yim, Yujuan Fu, Zhaoyi Sun, Meliha Yetisgen, Fei Xia, and Thomas Lin. 2024. Medec: A Benchmark for Medical Error Detection and Correction in Clinical Notes. *arXiv preprint* arXiv:2412.19260 (2024).

[2] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. 2025. Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 221–232. doi:10.1109/ICST62969.2025.10989033

[3] Danial Al. 2025. CLOC Repository. https://github.com/AlDanial/cloc Accessed 2025-09-29.

[4] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented Automated Test Case Generation. *Information and Software Technology* 176 (2024), 107565.

[5] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[6] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[7] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. Flakeflagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1572–1584.

[8] Gabin An, Juyeon Yoon, Thomas Bach, Jingun Hong, and Shin Yoo. 2024. Just-in-time Flaky Test Detection via Abstracted Failure Symptom Matching. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 741–752.

[9] Maider Azanza, Beatriz Pérez Lamancha, and Eneko Pizarro. 2025. Tracking the Moving Target: A Framework for Continuous Evaluation of LLM Test Generation in Industry. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. Association for Computing Machinery, New York, NY, USA, 524–534. doi:10.1145/3756681.3756946

[10] Thomas Bach, Artur Andrzejak, Changyun Seo, Christian Bierstedt, Christian Lemke, Daniel Ritter, Dong Won Hwang, Erda Sheshi, Felix Schabernack, Frank Renkes, et al. 2022. Testing Very Large Database Management Systems: The Case of SAP HANA. *Datenbank-Spektrum* 22, 3 (2022), 195–215.

[11] Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, Christoph Treude, and Stefan Wagner. 2025. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. arXiv:2508.15503 [cs.SE] https://arxiv.org/abs/2508.15503

[12] Alexander Berndt, Thomas Bach, and Sebastian Baltes. 2024. Do Test and Environmental Complexity Increase Flakiness? An Empirical Study of SAP HANA. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 572–581.

[13] Alexander Berndt, Thomas Bach, Rainer Gemulla, Marcus Kessel, and Sebastian Baltes. 2026. On the Flakiness of LLM-Generated Tests for Industrial and Open-Source Database Management Systems. https://doi.org/10.5281/zenodo.18148199

[14] Alexander Berndt, Sebastian Baltes, and Thomas Bach. 2024. Taming Timeout Flakiness: An Empirical Study of SAP HANA. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 69–80.

[15] Alexander Berndt, Zoltán Nochta, and Thomas Bach. 2023. The Vocabulary of Flaky Tests in the Context of SAP HANA. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–9.

[16] Matteo Biagiola, Gianluca Ghislotti, and Paolo Tonella. 2025. Improving the Readability of Automatically Generated Tests Using Large Language Models. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 162–173. doi:10.1109/ICST62969.2025.10989020

[17] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[18] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[19] SQLite Contributors. 2025. SQLite Docs. https://www.sqlite.org/ Accessed 2025-09-29.

[20] SQLite Contributors. 2025. SQLite Github Repository. https://github.com/sqlite/sqlite Accessed 2025-09-29.

[21] Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 982–994. doi:10.1145/3510003.3510194

[22] 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 (2024), 107468. doi:10.1016/j.infsof.2024.107468

[23] 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 (2024), 107468.

[24] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2024. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. arXiv:2408.11710 [cs.SE] https://arxiv.org/abs/2408.11710

[25] DuckDB Developers. 2025. DuckDB Documentation. https://duckdb.org/docs Accessed 2025-09-29.

[26] LangChain Developers. 2025. LangChain Docs. https://www.langchain.com/ Accessed 2026-01-03.

[27] MySQL Developers. 2025. MySQL GitHub Repository. https://github.com/mysql/mysql-server Accessed 2025-09-29.

[28] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 830–840. doi:10.1145/3338906.3338945

[29] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[30] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42. doi:10.1145/2685612

[31] Google. 2025. GoogleTest Docs. https://github.com/google/googletest Accessed 2025-09-29.

[32] Martin Gruber and Gordon Fraser. 2022. A Survey on How Test Flakiness affects Developers and What Support They Need to Address it. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 82–92.

[33] Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. 2024. Do Automatic Test Generation Tools Generate Flaky Tests?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[34] Mark Harman and Phil McMinn. 2009. A Theoretical and Empirical Study of Search-based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2009), 226–247.

[35] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18Th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 1–23.

[36] Mark Harman, Peter W. O'Hearn, and Shubho Sengupta. 2025. Harden and Catch for Just-in-Time Assured LLM-Based Software Testing: Open Research Challenges. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 1–17. doi:10.1145/3696630.3734199

[37] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 180–191. doi:10.1145/3696630.3728544

[38] Katherine L. Hermann, Hossein Mobahi, Thomas Fel, and Michael Curtis Mozer. 2024. On the Foundations of Shortcut Learning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=Tj3xLVuE9f

[39] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (2016), 122–131. doi:10.1145/2902362

[40] Minh Hoang and Adrian Berding. 2024. Presubmit Rescue: Automatically Ignoring FlakyTest Executions. In *Proceedings of the 1st International Workshop on Flaky Tests, FTW 2024, Lisbon, Portugal, 14 April 2024*. ACM, 1–2. doi:10.1145/3643656.3643896

[41] Dong Huang, Jie M. Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. 2025. Benchmarking LLMs for Unit Test Generation from Real-World Functions. *CoRR* abs/2508.00408 (2025). arXiv:2508.00408 doi:10.48550/ARXIV.2508.00408

[42] Neetha Jambigi, Thomas Bach, Felix Schabernack, and Michael Felderer. 2022. Automatic Error Classification and Root Cause Determination While Replaying Recorded Workload Data at SAP HANA. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 323–333.

[43] Konstantinos Kitsios, Marco Castelluccio, and Alberto Bacchelli. 2025. Automated Generation of Issue-Reproducing Tests by Combining LLMs and Search-Based

Testing. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, November 16 - November 20, 2025*.

[44] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3753–3766.

[45] Laurens Kuiper and Hannes Mühleisen. 2023. These Rows are Made for Sorting and That's Just What We'll Do. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2050–2062.

[46] Wing Lam, Kivanç Muslu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1471–1482. doi:10.1145/3377811.3381749

[47] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 312–322.

[48] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. 2024. A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites. arXiv:2408.07846 [cs.SE] https://arxiv.org/abs/2408.07846

[49] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 168–172. doi:10.1145/3510454.3516829

[50] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 643–653.

[51] Mateusz Machalica, Wojtek Chmiel, Stanislaw Swierc, and Ruslan Sakevych. 2020. How do you Test your Tests? https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/ Accessed 2025-09-17.

[52] Mateusz Machalica, Wojtek Chmiel, Stanislaw Swierc, and Ruslan Sakevych. 2020. Test Flakiness - One of the Main Challenges of Automated Testing. https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html?utm_source=chatgpt.com Accessed 2025-09-27.

[53] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 233–242.

[54] Alex Monahan. 2025. Vertical Stacking as the Relational Model Intended: UNION ALL BY NAME. https://duckdb.org/2025/01/10/union-by-name.html#:~:text=If%20only%20using%20the%20keyword,stacking%20occurs%20without%20additional%20processing Accessed 2025-09-29.

[55] Hannes Mühleisen and Mark Raasveldt. 2025. Runtime-Extensible Parsers. In *Proc. CIDR*.

[56] OpenAI. 2024. Gpt-4o System Card. *arXiv preprint arXiv:2410.21276* (2024).

[57] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-determinism of Chatgpt in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28.

[58] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[59] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. Multi-language Unit Test Generation using LLMs. *CoRR* abs/2409.03093 (2024). arXiv:2409.03093 doi:10.48550/ARXIV.2409.03093

[60] Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. Assessing AI Detectors in Identifying AI-generated Code: Implications for Education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. 1–11.

[61] Owain Parry, Gregory Kapfhammer, Michael Hilton, and Phil McMinn. 2025. Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures. *arXiv preprint arXiv:2504.16777* (2025).

[62] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[63] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2023. Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models. *Empirical Software Engineering* 28, 3 (2023), 72.

[64] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proc. ACM Softw. Eng.* 2, FSE (2025), 2897–2919. doi:10.1145/3729398

[65] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[66] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. 2024. The Effects of Computational Resources on Flaky Tests. *IEEE Trans. Software Eng.* 50, 12 (2024), 3104–3121. doi:10.1109/TSE.2024.3462251

[67] Ruixiang Tang, Dehan Kong, Longtao Huang, and Hui Xue. 2023. Large Language Models Can be Lazy Learners: Analyze Shortcuts in In-Context Learning. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 4645–4657. doi:10.18653/v1/2023.findings-acl.284

[68] Mistral AI Team. 2024. Mistral Large Instruct 2407. https://huggingface.co/mistralai/Mistral-Large-Instruct-2407 Accessed 2025-09-17.

[69] Stefan Wagner, Marvin Muñoz Barón, Davide Falessi, and Sebastian Baltes. 2025. Towards Evaluation Guidelines for Empirical Studies Involving LLMs. In *IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering, WSESE@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*. IEEE, 24–27. doi:10.1109/WSESE66602.2025.00011

[70] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Software Eng.* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[71] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A framework for detecting and fixing python order-dependent flaky tests. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 120–124.

[72] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *Proceedings of the 44th International Conference on Software Engineering*. 1730–1742.

[73] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Trans. Mach. Learn. Res.* 2022 (2022). https://openreview.net/forum?id=yzkSU5zdwD

[74] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 560–573. doi:10.1145/3676641.3716022

[75] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 50–61.

[76] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 481–503.