# Addressing Test Flakiness: Practical Approaches in a Database-Reliant Industrial System

George Vegelien*
georgevegelien@gmail.com
Delft University of Technology
The Netherlands

Carolin Brandt
c.e.brandt@tudelft.nl
Delft University of Technology
The Netherlands

Bas Graaf
bas.graaf@exact.com
Exact
The Netherlands

Arie van Deursen
arie.vandeursen@tudelft.nl
Delft University of Technology
The Netherlands

## Abstract

Flaky tests—tests that pass or fail unpredictably even without code changes—undermine the speed and trustworthiness of modern, database-heavy software systems. This study investigates the underlying causes of flakiness at Exact, a large-scale industrial system in which shared database states and resource contention introduced frequent test instability. By repeatedly rerunning the same commit, we pinpointed recurring problem areas and implemented three targeted solutions: (1) minimizing redundant background database tasks, and two strategies for cleaning up "dirty" tests: (2) explicitly disposing of test data and (3) disabling polluting tests with a database sanity check. Together, these interventions raised Exact's chance of a passing pipeline run from 27% to 95% and boosted their monthly release rate from 60% to 96%. Our findings confirm that rich, systematic measurement and well-prioritized fixes can significantly reduce flakiness in industrial-scale software.

## CCS Concepts

• **Software and its engineering** → **Maintaining software**; **Software testing and debugging**.

## Keywords

Flaky Tests, Measuring Flakiness, Software Testing, State Polluting Tests, Database-reliant Systems, Industrial Software Development

**ACM Reference Format:**
George Vegelien, Carolin Brandt, Bas Graaf, and Arie van Deursen. 2026. Addressing Test Flakiness: Practical Approaches in a Database-Reliant Industrial System. 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, 11 pages. https://doi.org/10.1145/3786583.3786891

*Work done during a thesis internship at Exact


*ICSE-SEIP '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786891

## 1 Introduction

Flaky tests, i.e., tests that pass or fail without any actual change in the code under test, have long been recognized as a significant challenge in software engineering [4, 8, 16]. They disrupt continuous integration (CI) and continuous delivery pipelines by generating false alarms and eroding developers' trust in test outcomes [4, 5]. If left unaddressed, flakiness encourages the habit of repeatedly rerunning tests instead of fixing the underlying issues, wasting substantial computational resources [18, 19] and further complicating already fast-paced deployment cycles [13].

Despite notable efforts in academia and industry to address test flakiness [22, 26, 28, 31, 32], this phenomenon persists, particularly when tests rely on external resources such as databases [13, 16]. Database-reliant systems often involve complex schemas, large amounts of shared data, and background maintenance tasks, all of which increase the unpredictability of test behavior [13, 21]. One test may pollute the database by leaving behind modified records, leading another test to fail without an obvious link to the change in data. Another source of instability arises from system tasks that run in parallel, causing state transitions or timeouts at seemingly random intervals. These issues become especially acute in industrial-scale systems with tens of thousands of tests, where a single sporadic failure can block a release pipeline [4, 13, 17].

In this paper, we take a closer look at test flakiness within Exact[1], a multinational software company with over 2,000 employees, empowering more than 650,000 customers with its business solutions. Their main product is a cloud-based software-as-a service business application for a variety of applications, such as payroll or human resource management. Built on the .NET platform, Exact heavily relies on a robust database infrastructure, currently utilizing hundreds of SQL databases, each containing upwards of 1,500 tables. This makes Exact a relevant context to investigate database-driven flakiness. Exact's continuous integration pipeline runs upwards of 25,000 tests hundreds of times daily across multiple platforms, aiming for frequent releases. Prior to our study, a considerable number of builds failed even when failing test were retried up to three times; most of these builds succeeded on a subsequent full pipeline rerun, indicating that the underlying causes might be environmental fluctuations rather than genuine product defects.

To investigate whether we can address test flakiness at Exact, we **adopt a measurement-driven approach** that repeatedly executes

[1] https://www.exact.com

the industrial test suite on the same commit and configuration. By gathering multiple pass/fail results for each test under identical conditions, we pinpoint tests that exhibit unpredictable outcomes. This process yields a granular view of both system-level flakiness (e.g., the likelihood that a given build will eventually pass all tests) and individual test flakiness (e.g., pass rates across repeated attempts). Drawing on insights from this aggregated data retrieved at Exact, **we then implement specific technical and organizational interventions that reduce flakiness**, such as minimizing redundant background tasks, improving test data handling via explicit disposal, and identifying or disabling "dirty" tests that pollute the database state. Leveraging our measurement-driven approach, we **quantify the impact of our interventions on the test flakiness** at Exact. Finally, we report on the overall impact that our study of flakiness had on the software at Exact and our understanding of flakiness measurement.

By focusing on practical mitigation strategies and embedding them within Exact's existing release process, we arrive at cost-effective interventions that can substantially reduce delays and increase release confidence. While *test-order dependency* is known to be one of the leading causes of flakiness [4, 9, 16], our results indicate that database reliance can compound flakiness with environmental factors such as external scheduling and resource contention, further underlining the importance of specialized solutions.

Concretely, this paper contributes:

- A new way of measuring relevant flakiness in an industrial system;
- Three automatic approaches to address flakiness in database-reliant systems;
- Empirical evidence on the impact of measuring and addressing flakiness in the industrial context at Exact.

This study provides actionable insights for both practitioners and researchers. These are deduced from our practical investigation into flakiness at Exact, and from the necessity to find the dominant and most impactful flaky test root causes. We believe the insights are not limited to Exact: our paper utilizes the scenario at Exact to demonstrate the benefits of adjusting the common conception regarding flakiness. In particular, we show that test flakiness in a database-heavy industrial system is not merely an issue of faulty test code or test order randomness. Instead, **flakiness often emerges from the interplay between infrastructure, shared state, and environmental variability**. Thus, we recommend quantifying flakiness using meaningful metrics, aggregating test results to find recurring patterns, and **adopting nuanced flakiness labels** rather than binary classifications. We stress the need to **define acceptable test pollution** boundaries to better identify and fix disruptive tests. Furthermore, for the research community, we call for a **distinction between impactful and negligible flakiness** and urge that benchmarks be run on realistic, production-like environments to ensure meaningful conclusions. These insights are grounded in over a year of production data and multiple targeted interventions.

In the following, we provide an overview of the background on flakiness and its known root causes (Section 2), then describe our measurement-driven approach (Section 3) and highlight the rationale behind each proposed intervention (Section 4). We present our results in terms of improved stability (fewer spurious CI failures) and quantifiable savings in developer time (Section 5). Our discussion demonstrates that large gains can be realized once an organization has clear visibility into how, when, and why flakiness occurs (Section 6). By combining automated detection and targeted fixes, our study aims to offer both researchers and practitioners a structured pathway for tackling flaky tests in database-centric industrial contexts.

## 2 Background and Related Work

### 2.1 Industry Context at Exact

Exact operates a large-scale automated test suite consisting of over 25,000 API and Integration tests, written in C# and VB.NET as well as Gherkin for selected scenarios[2]. The tests are executed on a hybrid infrastructure: Part of the test runs are executed on Exact's self-managed Azure DevOps agents hosted in private datacenters, while the load peaks are distributed to AWS agents in the cloud. All tests are executed during every Feature Regression Test (FRT) and Release Regression Test (RRT). There is no test selection based on code changes. Parallelization is achieved by distributing tests in chunks with a first-come-first-serve queue based system distributed over 2–5 virtual machines per stage, each with its own dedicated database instance templated to resemble production environments. To ensure isolation, tests are run sequentially per virtual machine, and database templates are populated with multiple user archetype instances to prevent data collisions and minimize test setup costs.

To mitigate the impact of flaky tests, Exact applies a strict 3-Attempt Retry Strategy. Each test starts with a first attempt. If it fails, the test is re-queued and executed up to two more times. If any of the three attempts passes, the test is labeled Passed on Retry. Otherwise, it is marked as Failed. This retry mechanism is applied consistently across both feature (FRT) and release (RRT) pipelines. FRTs are triggered on code merge requests and executed with standard resources. RRTs are run multiple times per day on higher-capacity infrastructure.

The RRT outcomes determine which builds are eligible for daily 4:00 a.m. production releases. This system ensures high release frequency while tolerating transient test failures that would otherwise slow down development. Nevertheless, flaky tests in a release regression test can block moving a build to production, leading to rework in examining the cause and delayed deployment of valuable functionality. It is the objective of this paper to identify ways to mitigate these undesired effects of flaky tests.

### 2.2 Related Work

Over the last decade test flakiness has become a well-documented impediment to reliable continuous integration and delivery. Systematic reviews identify four recurrent triggers: test-order dependency, test data pollution, resource contention, and environment variability [22, 26, 31].

*Order dependency*—the tendency of a test to pass or fail depending on the sequence in which it is executed—can be NP-hard to expose and is estimated to account for 8–60% of all flaky failures, with higher prevalence in dynamically typed languages [9, 14, 30].

[2] https://cucumber.io/docs/gherkin/

*Test data pollution* denotes any residual state left behind by one test that influences subsequent tests; polluted database records or global configuration flags have been shown to undermine CI reliability in industrial studies by Bell, Marinov and others [11, 15, 21]. In our study, the flakiness interventions described in Section 4.2 consider test data pollution and through this indirectly test order-dependency.

*Resource contention* encompasses concurrency issues such as thread scheduling, network latency or database locks. Lam et al. demonstrated that seemingly harmless shifts in computational resources can alter failure rates in otherwise deterministic code [25].

Lastly, *environment variability* refers to differences in hardware, operating-system version or virtualisation layer that surface non-deterministic behaviour only in specific configurations, a problem repeatedly observed at large providers such as Google, Mozilla and others [4, 13, 19, 21]. Our intervention in Section 4.1 targets resource contention, and throughout the whole study we also observe an impact of execution environment changes on flakiness.

Detecting flakiness typically relies on controlled reruns, static or dynamic heuristics, or a combination thereof. Frameworks such as *iDFlakies* and *iPFlakies* repeatedly execute suites under shuffled orders to reveal order-dependent failures [14, 29]. Google's internal entropy-based flakiness score depends on many reruns but scales to millions of tests in nightly infrastructures [12, 19]. In contrast, learning-based tools including *FlakeFlagger*, *DeFlaker* and *NonDex* attempt to predict or provoke flakes without exhaustive repetition, exploiting change coverage, stack traces or non-deterministic API stubs [1, 3, 10]. While reruns give statistically sound pass-rate estimates, they are computationally expensive. Tools working with learned heuristics trade accuracy for speed. For our case at Exact, precise measurement of system-relevant flakiness was crucial. This is why we apply a rerun-based approach.

Typically, there are three lines of attack to reduce flakiness. First, many organisations retry or quarantine failing tests, an expedient practice that masks deeper defects and escalates infrastructure cost [19, 21]. Second, automated transformations—exemplified by *iFixFlakies*—rewrite tests or reorder statements to break hidden dependencies [24]. Third, resource isolation through containers or in-memory databases eliminates interference at the price of replicating complex production schemas [2, 6, 7, 20]. To date, comparatively little research examines how background maintenance tasks in enterprise databases interact with these strategies; Lam et al.'s *RootFinder* remains one of the few studies in this space [13].

The evidence from industrial reports underscores the economic stakes of combatting flakiness. Case studies at Google, Apple, Facebook, Ericsson and Mozilla attribute release delays, costly reruns and developer frustration directly to flaky pipelines [12, 19, 23]. Yet, most published datasets centre on unit-heavy Java projects or mobile UI suites. Database-centric back-ends with large shared schemas and background jobs are under-represented, leaving open questions about the relative contribution of resource contention and data pollution in such environments.

**Gap addressed.** Our study extends the empirical knowledge on test flakiness by analyzing approximately 25,000 API and integration tests in a production .NET/SQL stack at Exact. We show that redundant background database tasks and cross-test data pollution are dominant triggers: by rerunning identical commits on production hardware we narrow down actual flakiness, including environment-induced failures. We then apply lightweight countermeasures including task suppression (Section 4.1) as well as explicit disposal and sanity checks (row-count and configuration-hash, Section 4.2). The resulting stability gains demonstrate the value of database-specific interventions in large industrial CI pipelines.

## 3 Measuring Flakiness

We propose a measuring regime for test flakiness that helps the development team identify and address the most critical instances of flakiness, and assess the progress over time in reducing test flakiness.

Our starting point is that we measure flakiness by repeatedly rerunning the full test suite *for a given commit*. This yields a focus on flakiness caused not by change, but by environmental factors or hidden state changes as they might happen in database-centric systems.

### 3.1 Pipeline Pass Probability

Key concepts for our metric, called the **Flaky Pipepline Pass Probability**, are shown in Figure 1.

Each individual execution of a test method is called a **test attempt**. A test is allowed up to three attempts: the initial run and up to two retries if it fails. These attempts together form a **test run**, which passes as soon as an attempt succeeds. A **pipeline run** consists of thousands of such test runs executed in parallel. The pipeline passes only if all test runs pass. The **pass rate** of a test is defined as the fraction of test attempts that pass.

A **benchmark run** refers to a collection of pipeline runs executed on the same code version (same commit), used to evaluate the stability and flakiness of tests under controlled conditions. The **Flaky Pipeline Pass Probability (FPPP)** is computed as the percentage of pipelines within a benchmark that pass. In Figure 1, the full pipeline is run three times on the same benchmark commit. Two out of these three pipepline runs have only passing tests after up to three attempts, but the third has test runs where all three attempts fail, leading to an FPPP of 66%.

Clearly, higher pass rates are preferable, as they provide more stability and confidence. As we will see, thanks to targeted interventions, over a period of months the FPPP at Exact climbed from 27% to 96%.

### 3.2 Periodic Error Grouping

To support the identification of flakiness *causes* and potential *interventions* to resolve such causes, we periodically (around once per month) conduct a pipeline analysis with a focus on the errors its failures produce.

Here again, we run the entire test suite multiple times on a single commit (typically around 30 times), on infrastructure that is as close as possible to Exact's production environment. We then log each pass/fail attempt along with its error messages and systematically group these messages using string and stack-trace similarities. This manual clustering helps us to detect patterns beyond isolated failures. For example, a large number of timeouts could indicate systemic resource contention, while repeated "access denied" errors suggest misconfigured authentication or permissions. By mapping

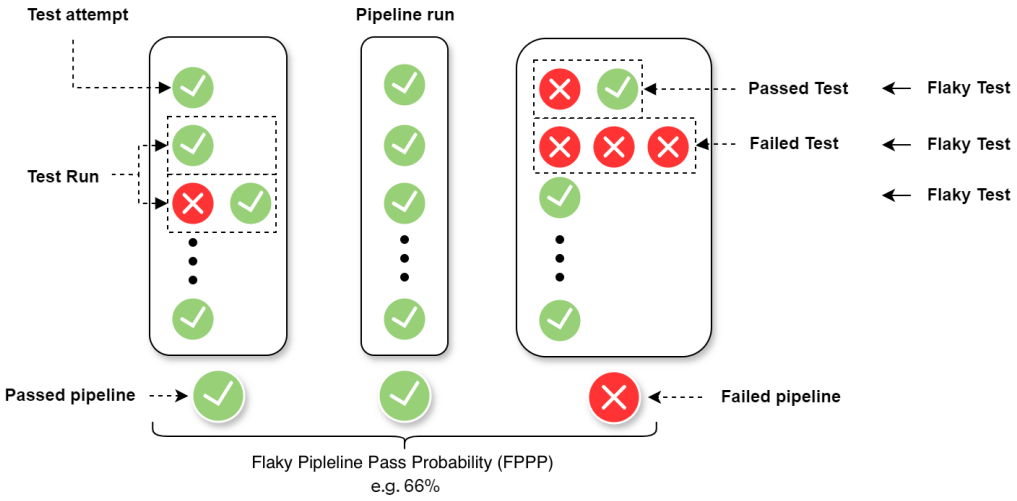


**Figure 1: Definitions of test attempt, test run, pipeline run, and flaky test, illustrating the 3-attempt policy and Flaky Pipeline Pass Probability (FPPP).**

recurring error signatures to specific code paths or subsystems, we gain early insights into which segments of the platform disproportionately contribute to flakiness. This analysis, conducted together with Exact engineers, has pointed us to the causes and interventions covered in Sections 4.

### 3.3 Sporadic Flakiness

To be able to prioritize the most urgent causes of flakiness, we distinguish between *sporadic* and *frequent* flaky tests. We refer to a flaky test case that fails in less than 10% of the runs as **sporadic** flakiness, while a flaky test that fails in more than 10% of the runs as **frequent** flakiness.

Our assumption is that sporadic flakiness may be related to transient effects such as minor race conditions or infrastructure failures. A cost-effective solution mitigation may be test re-runs or short-term refinements (like short extra timeouts).

Frequent flakiness, by contrast, may point to deeper structural issues such as ongoing database background tasks or improperly handled data states. These warrant extra investigation, and may necessitate more substantial changes to the database configuration or test logic. The interventions proposed in this paper address such frequent flakiness.

## 4 Addressing Flakiness

Measuring test flakiness (Section 3) and grouping based on flaky error signatures provides a foundation for spotting unstable areas.

Our data revealed two dominant contributors in a database-dependent context: **excessive background tasks** and **polluted test data**. Together, these two root causes explained 62% of all pipeline-blocking failures (40% and 22% respectively), i.e., failing Release Regression Tests (Section 2.1) that blocked release to production. Together, these root causes were responsible for a Flaky Pipeline Pass Probability that was as low as 27%. Although other forms of flakiness appeared (including timing issues and race conditions), these two categories accounted for the majority of test failures observed in repeated runs.

Below we describe targeted flakiness interventions that address these root causes: minimizing background tasks and addressing test poulltion through explicit disposal and database sanity checks.

### 4.1 Minimizing Database Background Tasks

Our first flakiness intervention reduces the automated background behaviour during tests by modifying the database configuration and its data. Background tasks—such as scheduled clean-ups or watchers—were present at Exact because the team historically duplicated the entire production environment inside every transient test database. Since the database exists only temporarily for the duration of the test run, these background tasks add overhead without functional value. By reevaluating the test database configuration, we isolate the testing environment from incidental slowdowns or data inconsistencies. Our hypothesis is that fewer concurrent database operations lead to more deterministic test outcomes, especially for frequent flaky tests that exhibit high failure rates.

### 4.2 Addressing Test Pollution

To reduce test pollution, we adopted a proactive and a reactive strategy: The *proactive* intervention introduces **explicit disposal**: our WeDispose refactoring inserts deterministic `dispose()` calls to dispose of created test data within a test. We used static analysis to identify and refactor all tests that did not dispose of their test data. Figure 2 illustrates how we explicitly disposed of variables and how not disposing can lead to other tests failing. This practice of disposing of the test data is expected by Exact's internal framework culture, but not formally enforced.

In the *reactive* approach, we identified existing polluting tests by running a series of database sanity checks at two levels: A *row-count* check and a *configuration-hash* check. These checks compared the database state before and after execution of each test, surfacing

```
// Reuses live connection to increase performance.
Connection conn;  \\ IDisposable
public void activateConnection() {
    if (conn is null) {
        conn = new  dBConnection();
    }
    if (conn.isActive) return;
    conn.poolConnections();
    conn.start();
}

[TestMethod]
public void TestA()
{
   conn = new MockConnectionForTestA(...);
   ...
+   conn.dispose();
}

[TestMethod]
public void TestB()
{
   activateConnection();
   // Test will fail if still using mock conn of testA,
   // i.e. when executed after testA without conn.dispose().
   ...
}
```

**Figure 2: Adjusting `TestA` by adding a `dispose` to make sure `TestB` will not fail.**

inconsistencies or residual data that could affect subsequent tests. Both checks are implemented as shown in Figure 3, by injecting a `saveState` method before each test and a `compareState` method after it. The specific implementation of these methods differs depending on whether the row-count or configuration-hash check is used. While the row-count check widely targets all data, the configuration-hash check was developed together with Exact's engineers. It focuses on specific tables that they identified as relevant for basically all operations, so likely modified by a test and impacting another test if not correctly set. To measure the flakiness impact, we then disabled all tests that were flagged as polluting by the sanity checks in one full test suite run.

## 5 Results

We deployed each of the interventions from Section 4 iteratively at Exact and tracked the impact using our measurement protocols outlined in Section 3. This section details how flakiness rates changed after the interventions were put in place. Concretely, we address the following research questions:

**RQ1:** What is the impact of measuring on test flakiness at Exact?
**RQ2:** How does minimizing DB background tasks impact test flakiness at Exact?
**RQ3:** How does addressing test pollution impact flakiness at Exact?

```
[TestInitialize]
public override void TestInitialize()
{
// Saves the current DB row counts/hashed settings
+   SanityCheck.saveState();

    base.TestInitialize();
   ...
}

[TestCleanup]
public override void TestCleanup()
{
    ...
    base.TestCleanup();

// Compares the saved values to the current ones
+   SanityCheck.compareState();

}
```

**Figure 3: Adding database sanity checks to test setup and teardown.**

RQ1 quantifies our baseline by asking whether measurement alone changes developer behaviour; RQ2 isolates concurrency effects from background tasks; and RQ3 focuses on cross-test data contamination revealed by our two sanity checks.

To address these questions, over a period of nine months, we measured flakiness at Exact and iteratively deployed our interventions. The results are presented below.

### 5.1 (RQ1) Flakiness Patterns and Improvements Over Time

To understand the prevalance and trends of flakiness, we conducted a series of flakiness measurements. Each measurement consisted of 26 to 32 reruns of all tests for the same commit, on Exact's production test hardware.

*Sporadic vs. Frequent Flaky Failures.* Figure 4 shows the prevalence of flaky failures in ten bins of 10% pass rate each. The figure distinguishes *failed* tests (which lead to three failures in a row after three retries, orange in the figure) from *flaky* tests (which have one failure but which end up in a pass after one or two retries, blue in the figure):

- The *sporadic* bin of (0.9,1) is the largest, being responsible for 74.9% of the flaky tests. They lead, however, only to 27.5% of the failed tests that blocked a pipeline: As they are sporadic, a retry often resolves the issue.
- The remaining nine *frequent* (non-sporadic) bins, by contrast, total to only 25.1% flaky tests, but they cause 72.5% of the failing test attempts. This, in turn, results in 69% of all pipeline failures. Clearly, retrying here does not help.

Making these percentages available to the Exact developers inspired them to investigate the underlying causes. A substantial fraction of the sporadic failures resulted from external infrastructure factors, such as offline services or resource contention across virtual machines. Changes to the test platform also caused instability in some tests (e.g., differing configuration or hardware setups),

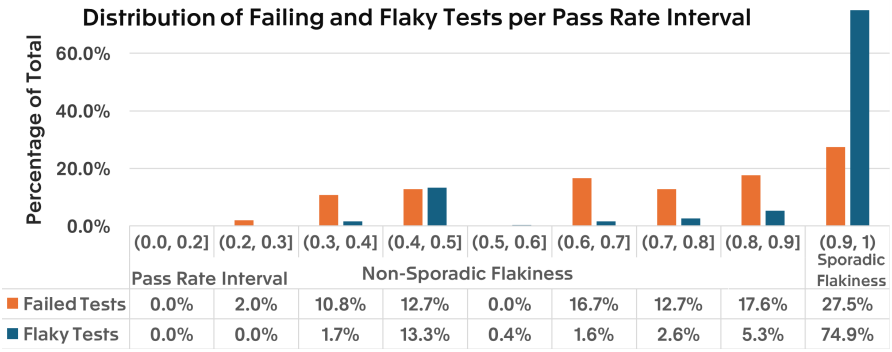


**Figure 4: Distribution of failing (causing at least one pipeline run to fail through three failed attempts) and flaky tests (with at least one failing attempt over the whole benchmark) per pass rate interval for all gathered benchmarks. This shows that considering tests falling under non-sporadic flakiness (<= 0.9 pass rate), are much more likely to actually lead to pipeline failures.**

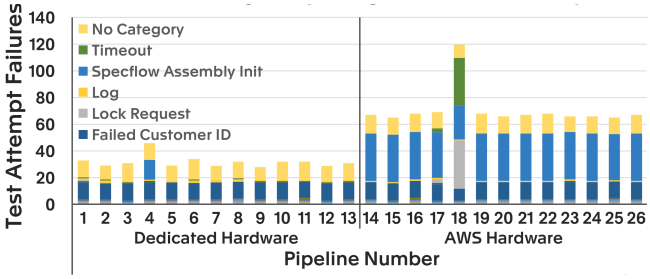


**Figure 5: Flaky tests for 26 pipelines in six different categories, with halfway a switch in infrastructure affecting flakiness.**

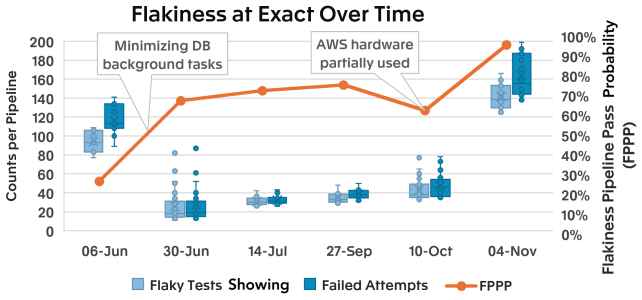


**Figure 6: Flaky tests at Exact over time, gathered from baseline benchmark runs. Note: FPPP represent how often both the API and Integration test stage passes when rerun on the same release with the same configuration.**

occasionally affecting random subsets of tests as well as a specific subset linked to hardware dependencies. By contrast, frequent (non-sporadic) failures trace back to persistent problems, within the code or within database configurations.

Another perspective on sporadic versus frequent flaky failures is shown in Figure 5. Here test failures are grouped into six categories obtained through manual analysis of the flaky tests, such as recurring issues related to timeouts, incorrect customer IDs, or the initialization of the SpecFlow engine (for Gherkin-based tests). The figure shows a total of 26 pipelines, with halfway a switch in hardware. On dedicated hardware (runs 1–13), the mix of failure types is broadly similar from run to run. However, a noticeable bump of approximately fifteen extra *SpecFlow Assembly Init* errors appears in run 4. When the same tests move to AWS (runs 14–26), the base distribution persists, but a *constant* block of additional *SpecFlow Assembly Init* failures now appears in every pipeline. Again, one run (no. 18) experiences an isolated spike of extra *Timeout* and *Lock Request* errors. The consistent distribution of failures on most runs represents the non-sporadic failures, while the outliers in runs 4 and 8 point to sporadic failures.

Together, Figures 4 and 5 underscore why this distinction between both groups matters: non-sporadic tests dominate long-term risk, while sporadic, environment-driven bursts can suddenly inflate failure counts and conceal deeper issues.

*Gradual Improvement in Pipeline Stability.* Figure 6 tracks the Flaky Pipeline Pass Probability (FPPP) over the duration of our study. At each data point we shared flakiness reports with the company. The vertical box plots compare "Flaky Tests Showing" (the number of tests failing at least once) against "Failed Attempts" (all individual failed attempts) per pipeline run. This indicates how many tests are flaky per pipeline on average. The difference between the boxplots indicates how many fail on second and third attempts as well. In terms of FPPP, the beginning is marked by a pronounced increase from the 27% of pipeline pass probability following the initial reporting of flakiness and flaky tests on June 6. Initially, a notable gap existed between flaky tests and failed attempts, indicating tests that repeatedly failed within the same run. By June 30, the difference largely disappeared, showing a decrease in repeatedly failing test attempts. However, after October 10, a new hardware environment in AWS coincided with an uptick in both sporadic and repeated failures. The FPPP remained high over the later phases of our study, up to 96% on the last measurement on November 4. However, the later data showed the discrepancy

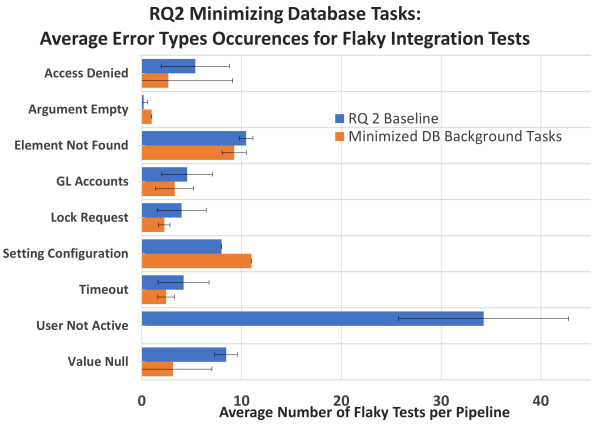


**Figure 7: The average distribution of flaky tests observed within RQ 2 flaky-test benchmarks, categorized based on error messages and failure types. Note: A single flaky test may be classified under multiple categories if it exhibits error messages that fall under multiple categories or if it experiences multiple failed attempts with different error messages.**

between "Flaky Tests Showing" and "Failed Attempts" gradually widening, suggesting a possible future decline in pipeline reliability if left unaddressed. This widening coincided with less frequent sharing of flakiness reports.

*Release Rate Improvements.* In addition to raising the FPPP, Exact's monthly release rate increased considerably after attention turned to mitigating flaky tests. In May prior to our study, the percentage of targeted release days at which a successful release took place—which we will call the *successful release rate*—had dipped to a low of 60%. After we started sharing flakiness reports, the successful release rate rose to 95% for three consecutive months beginning in July. Company metrics suggest this outcome was partly attributable to the same-commit benchmark reports, which enabled developers to identify and fix the most problematic tests before they triggered repeated release-blocking failures. Although new flaky tests continued to appear, many now passed on subsequent attempts within the same pipeline run and thus posed fewer immediate obstacles to successful deployments.

**Findings for RQ1: What is the impact of measuring on test flakiness at Exact?**
Repeated same-commit benchmarks uncovered that one quarter of the flaky tests (the non-sporadic cluster) caused nearly 70% of pipeline failures. Sharing these measurements motivated focused efforts within Exact to address flakiness.

## 5.2 (RQ2) Raising Test Stability by Suppressing DB Background Tasks

Figure 7 shows a distribution of flaky tests over nine categories, which emerged from discussions with Exact engineers. Investigation of the "User Not Active" category revealed a potential connection with background database tasks, leading to the intervention described in Section 4.1.

The blue bars indicate prevalance of the categories before applying the intervention; the orange bar the prevalence after. The "User Not Active" category completely disappeared, and the "Value Null" category was substantially reduced. Overall, this intervention yielded a 1.6-fold improvement in the Flaky Pipeline Pass Probability (FPPP), from 27% to 44%, and lowered the median count of flaky tests per pipeline by about 40%.

A closer breakdown of test-level pass rates indicates that 60 of the original 115 non-sporadic flaky tests (pass rate < 0.9) now pass consistently. Most of these previously failed with one of the highest-frequency failure categories that hinted at resource contention. Ten new tests became non-sporadically flaky. Manual debugging revealed that these failures were attributable to test data pollution instead of a dependency on the background tasks that we minimized with the intervention. Finally, 41 of the 115 non-sporadic flaky tests remained unchanged. No direct adverse consequences (e.g., undersimulation of production) were observed, suggesting that removing unnecessary background operations can reliably improve stability in database-centric test environments.

**Findings for RQ2: How does minimizing DB background tasks impact test flakiness at Exact?**
Disabling redundant background tasks eliminated two high-frequency error categories ("User Not Active" and "Value Null"), fixed 60 of the 115 non-sporadic flaky tests, and produced a 1.6x improvement in FPPP without observed negative side-effects on functional representativeness of the test suite.

## 5.3 (RQ3) Addressing Test Pollution

Our final research question addresses the impact of the interventions aimed at addressing pollution, as laid out in Section 4.2: (1) adding missing explicit disposal of test data, and (2) detecting and disabling tests that pollute shared database state. Both approaches reduced certain forms of flakiness, but also uncovered new brittle dependencies.

*Explicitly Disposing Data.* Our static analysis-powered refactoring introduced explicit `dispose()` calls in 5,064 tests, revealing that over 20% of the entire 25k integration and API test corpus created test objects that were not properly cleaned up. Our measurements indicated that by explicitly disposing, only 5 frequently failing tests stopped exhibiting flakiness. After manual inspection, we could not identify root causes for their flakiness. No explicit dispose calls were added within their test class. This indicates that their flakiness likely stemmed from leftover open connections or database pollution.

On top of this some new, now consistent test failures emerged. In particular, 27 brittle tests [24] began failing consistently, as they relied on other tests to establish necessary data. This also included

inter-assembly coupling, where a disposable object was removed in a different class, leading to missing data or incomplete setups triggering consistent test failures.

*Database Sanity Checks (Dirty Tests).* Our two sanity checks—a general row-count monitor and the specialized configuration-hash check—flagged 11.43% of tests as modifying the database state without restoring it. Disabling these flagged tests lead to 28% of the non-sporadically flaky tests no longer being flaky. Disabling these flagged tests entirely prevented 11 previously non-sporadic failures but caused 5 always-passing tests to fail, indicating that those tests depended on leftover data from the removed polluting tests. Moreover, 2 tests became newly flaky for reasons unrelated to the disabled dirty tests, likely due to altered execution order. Although the generalized check identified more polluting tests than the specialized approach, the specialized check was more successful in reducing overall flakiness [27].

**Findings for RQ3: How does addressing test pollution impact flakiness at Exact?**
Both strategies, explicitly disposing test data and disabling database-polluting tests, underscore how hidden dependencies and data pollution can undermine test reliability. While the strategies each fixed some flaky behaviors, they also revealed brittle tests relying on shared state or inconsistent setup. In sum, test pollution in this large database-reliant system was significant yet nuanced, requiring targeted remediation that can inadvertently expose other dependencies if not carefully managed.

## 6 Discussion

In this section we discuss our overall findings and impressions from our nine month study investigating and tackling flakiness at Exact with targeted interventions. We note key limitations of our results and point to the next necessary steps for industrial flakiness research. Our takeaways focus on two main themes: the inherently multifaceted, combinatorial nature of flakiness in an industrial context and the importance of rich, aggregated information for effectively combating flakiness.

### 6.1 A Handful of Tests Drive Pipeline Instability.

Our measurement-driven approach revealed that a small subset of frequently failing tests caused a disproportionate share of pipeline disruptions. These "non-sporadic" flaky tests tended to arise from diverse causes, such as database background tasks, test pollution, or environment-dependent behavior. Moreover, many flakiness issues surfaced only when multiple problems co-occurred—e.g., one test leaving behind partially updated data, another relying on that data, and a background routine contending for resources. In several cases, making a "fix" in one area (like disposing of test objects more aggressively) inadvertently revealed or exacerbated a different dependency. This interplay underscores that flakiness is best understood as a *multifaceted* phenomenon: tests may fail non-deterministically for a variety of reasons, and seemingly unrelated changes can trigger new failures or expose hidden brittleness.

Furthermore, our benchmarks confirmed that flakiness *exhibits differently across test suites and environments.* Integration and API test had various different causes for flakiness, and while they where often related to subtle concurrency or database-related issues, they stemmed from various reasons and exhibited differently. Similarly, the use of AWS hardware versus dedicated internal machines created varying performance and timing conditions that amplified some classes of flakiness while reducing others. These observations reinforce that flakiness rarely has a one-size-fits-all solution; rather, teams must measure the flakiness in their system, adapt their strategies to the particular stage (UI vs. API vs. integration), environment (cloud vs. on-prem), or test design.

### 6.2 Aggregated Metrics Accelerate High-Impact Flakiness Mitigations.

A central takeaway for us from this study with Exact is that *presenting flakiness data in granular and summarized forms* helped both developers and management to act on the issues. High-level overviews (e.g., error categories and pass rates) exposed systemic faults—such as entire assemblies prone to "User Not Active" errors—rather than scattering them across many individual tickets. Once teams realized that, for instance, 40% of failures linked to a single background database routine, they quickly recognized the urgency of disabling it in the transient test database. Similarly, quantifiable metrics like the Flaky Pipeline Pass Probability (FPPP) made the problem's business impact visible to management, motivating resource allocation for deeper fixes. Without these aggregated insights, flakiness would likely persist in sporadic or uncorrelated bug reports, hampering targeted interventions.

At the same time, we observed an over-reliance on a blanket "flaky test" label, which can dilute the meaningful distinction between genuinely fragile tests and failures caused by rare or one-off events. A single pass/fail marker may mislead developers into ignoring infrequent but high-impact problems, or into quarantining valid tests due to incorrectly attributing them to "flakiness." This is why we opted to distinguish sporadic and the higher-priority non-sporadic flakiness. These findings align with prior industry observations [12] that *some form of continuous scoring or historical flakiness measurement* is more appropriate than a purely binary label, especially for large, long-lived systems. By monitoring pass rates, pipeline outcomes, and error messages over time, teams can distinguish severe, recurring flaky behaviors from sporadic or inconsequential failures and thus deploy their resources more effectively.

### 6.3 Recommendations and Implications

Based on our findings, we can make the following recommendations, to engineering teams, tool vendors, and academic researchers.

*Recommendations for Practitioners.* From our findings, we suggest that engineering teams:

- **Quantify flakiness** through clear metrics such as pass rates and FPPP. Showcasing which test are flaky and how flakiness affects build and release success helps align both developers and management.

- **Aggregate test results** to spot patterns, rather than investigating each failure in isolation. This uncovers root causes that may span multiple tests or assemblies.
- **Recognize environment-specific behaviors**, as flakiness can look very different in on-premise environments versus cloud-based ones.
- **Use continuous or gradual labeling** rather than a strict "flaky vs. non-flaky" dichotomy. Gradual scoring or recurrence-based categories help distinguish consistently disruptive tests from minor one-off issues.
- **Explicitly define which tests pollution is allowed**. Not all flaky tests indicate a problem with the test itself, nor do dirty tests always cause other tests to become flaky. Cleaning up this pollution then becomes a tradeoff. Teams should explicitly define acceptable versus unacceptable test pollution within their test suites. Establishing clear guidelines helps developers to adjust their tests accordingly allowing tools and developers to identify genuinely problematic tests (offending tests), facilitate automatic targeted fixes and will ultimately reduce wasted effort spent investigating falsely labeled flaky tests.

*Recommendations for Tool Vendors and Builders.* Some of the lessons learned from our study have implications not just for teams managing their test suites, but also for tool builders and vendors trying to help such teams.

Effectively addressing test flakiness requires not only automated metrics, but also developer interpretation and domain knowledge. Many of the techniques in this study depend on context-specific understanding—such as context clues for common flakiness reasons of test pollution or environment-related behavior—that cannot be resolved by automation alone. Making flaky test data directly visible within CI platforms allows teams to apply this domain knowledge where it matters. The aggregation methods and metrics used in this study are broadly applicable and can be generalized across CI/CD systems. If integrated into existing toolchains, they could empower teams across the industry to detect, interpret, and act on flakiness more effectively. Based on what proved most useful at Exact, we suggest the following functionality:

- CI platforms such as Azure DevOps, GitHub Actions, and GitLab CI should offer **native flaky test observability**, including key metrics such as per-test pass rates, Flaky Pipeline Pass Probability (FPPP), and retry-aware flakiness scores, making instability measurable by default.
- Flakiness data should be **automatically aggregated and visualized** across benchmark runs, environments, and test agents, enabling teams to prioritize consistently disruptive tests and monitor instability trends over time.

*Recommendations for Academics.* Lastly, the research community has expressed a strong interest in investigating flakiness, as covered in Section 2. Based on our findings, we suggest the following:

- **Distinguish "useful" flakiness.** Treat a test that fails once in 10,000 runs very differently from one that fails every other day; Research and engineering effort is spent more wisely spent on the latter than on the former.
- **Benchmark flakiness in production-like environments.** Attempting to reproduce flaky behavior on synthetic setups or reordered test sequences often yields misleading results. Instead, rerun benchmarks should be performed on production-like infrastructure with realistic resource constraints and test orderings to capture the true conditions under which flakiness occurs.

Overall, our findings illustrate that flakiness in a database-reliant industrial setting is neither a simple ""bug in the test" nor a purely external phenomenon. Its root causes are often combinational and can shift as tests, platforms, and production-like conditions evolve.

## 6.4 Threats to Validity

We identify the following threats to validity.

**Internal validity.** Our same-commit rerun benchmarks measure flakiness at a single revision; unseen code changes between monthly runs might silently introduce or mask flaky behaviour.

**Construct validity.** Error-message grouping relies on manually curated patterns; some root causes may therefore be under- or over-represented. The FPPP captures build-level stability but not developer frustration or time-to-fix.

**External validity.** Exact's database-heavy architecture and five-agent test fan-out may differ significantly from other organisations. Our results may generalize only to pipelines with comparable data volumes and background-task footprints.

**Reliability.** We repeated multiple pipeline runs over a day for each benchmark on production test hardware to curb incidental noise, yet sporadic infrastructure failures (e.g. transient cloud outages) can still influence individual data points.

**Data availability.** As this research is conducted in an industrial context, the underlying data and code cannot be made available openly.

## 6.5 Limitations and Future Directions

While our rerun-based data collection isolates flakiness at a particular code state, it may not capture ongoing changes in the codebase or environment. Large industrial codebases often undergo many changes within a short time-frame. In addition, repeated pipeline executions require significant computational overhead; organizations with extremely high release frequencies may instead choose to rely on partial reruns or historical time-series data. Another limitation is that our manually defined error-type categories, while effective at capturing local patterns, might not generalize without adaptation to other projects.

Looking ahead, future work could explore more granular or automated grouping of flaky tests using natural-language processing and stack-trace mining—e.g., to unify error messages that vary in detail but reflect the same underlying root cause. Another promising direction would be to integrate same-commit reruns with time-series flakiness scores, capturing both snapshot and historical perspectives. Furthermore, more research can be done regarding different test layers (UI vs. integration vs. unit) and how diverse hardware environments impact flakiness, which could help refine how best to prioritize and fix flakiness under varying conditions. Finally, future work could investigate the long-term effect of flaky test reports on software health, development speed, and test flakiness, since we

have seen many great improvements over only a shorter time-frame of a couple of months.

## 7 Conclusion

This study has shown that flaky tests are not mere technical nuisances; they systematically disrupt pipelines, erode developer confidence, and can substantially raise the cost of software development. To address these challenges in a large-scale database-centric environment, we make two key contributions. First, we advocate repeatedly rerunning the same commit to map out the most impactful flaky tests, and demonstrate the effectiveness of using the Flakiness Pipeline Pass Probability (FPPP) for measuring the level of flakiness. Second, we identify a number of interventions targeting root causes (e.g., redundant database background tasks, implicit object disposals, and database pollution) through automated analysis and selective disabling or refactoring of problematic tests.

Through our case study at Exact, we demonstrated how (1) reducing redundant database tasks lowered the average number of flaky tests by 40%, (2) explicit disposal rules addressed hidden coupling, and (3) database sanity checks uncovered polluting tests that created brittle interdependencies. Altogether, these measures lifted Exact's FPPP from 27% to 96%, facilitating a record-high release rate and a notable decline in wasted resources.

Crucially, our findings underscore that flakiness is both *multi-faceted* and *combinatorial*. Individual failures can arise from timing issues, data leaks, improperly disposed resources, or overlapping background tasks—and many of these factors only become problematic in combination. By rerunning the entire test suite on the same commit, aggregating detailed error messages, and categorizing high-frequency failure types, organizations can more effectively diagnose and remediate the underlying causes of flakiness. These results are broadly applicable, offering a concrete framework for measuring test instability, developing targeted fixes, and sustaining a more reliable continuous integration pipeline.

## Acknowledgments

This research was partially supported by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032) and conducted as part of George Vegelien's master thesis [27]. We thank Exact for supporting our research and especially all their engineers who shared their insights with us throughout this study.

## References

[1] 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)*. 1572–1584. doi:10.1109/ICSE43902.2021.00140

[2] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 550–561. doi:10.1145/2568225.2568248

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 433–444. doi:10.1145/3180155.3180164

[4] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. doi:10.1145/3338906.3338945

[5] Martin Fowler. 2011. *Eradicating Non-Determinism in Tests.* https://martinfowler.com/articles/nonDeterminism.html Accessed 30 Jan 2025.

[6] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 1–11. doi:10.1109/ICST.2018.00011

[7] Maxime Gobert, Csaba Nagy, Henrique Rocha, Serge Demeyer, and Anthony Cleve. 2023. Best practices of testing database manipulation code. *Information Systems* 111 (2023), 102105. doi:10.1016/j.is.2022.102105

[8] Google. 2008. *TotT: Avoiding Flakey Tests.* https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html Accessed 30 Jan 2025.

[9] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 148–158. doi:10.1109/ICST49551.2021.00026

[10] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 993–997. doi:10.1145/2950290.2983932

[11] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 223–233. doi:10.1145/2771783.2771793

[12] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 110–119. doi:10.1145/3377813.3381370

[13] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 101–111. doi:10.1145/3293882.3330570

[14] 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)*. 312–322. doi:10.1109/ICST.2019.00038

[15] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 202 (Nov. 2020), 29 pages. doi:10.1145/3428270

[16] 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* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. doi:10.1145/2635868.2635920

[17] John Micco. 2016. *Flaky Tests at Google and How We Mitigate Them.* https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html Accessed 30 Jan 2025.

[18] John Micco. 2017. The state of continuous integration testing@ Google. In *ICST*. https://www.aster.or.jp/conference/icst2017/program/jmicco-keynote.pdf Accessed 30 Jan 2025.

[19] John Micco and Atif Memon. 2016. *How Flaky Tests in Continuous Integration.* Google Inc. https://www.youtube.com/watch?v=CrzpkF1-VsA Google Test Automation Conference 2016.

[20] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 496–499. doi:10.1145/2025113.2025202

[21] Jason Palmer. 2019. *Test Flakiness – Methods for identifying and dealing with flaky tests.* Spotify. https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/ Spotfy R&D Engineering. Accessed 30 Jan 2025.

[22] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (Oct. 2021), 74 pages. doi:10.1145/3476105

[23] Maaz Hafeez Ur Rehman and Peter C. Rigby. 2021. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1371–1380. doi:10.1145/3468264.3473930

[24] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia)

*(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 545–555. doi:10.1145/3338906.3338925

[25] 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 Transactions on Software Engineering* 50, 12 (Dec 2024), 3104–3121. doi:10.1109/TSE.2024.3462251

[26] Amjed Tahir, Shawn Rasheed, Jens Dietrich, Negar Hashemi, and Lu Zhang. 2023. Test flakiness' causes, detection, impact and responses: A multivocal review. *Journal of Systems and Software* 206 (2023), 111837. doi:10.1016/j.jss.2023.111837

[27] George Vegelien, Arie Deursen, Carolin Brandt, and Bas Graaf. 2025. Addressing Test Flakiness: Practical Approaches in a Database-Reliant Industrial System. https://resolver.tudelft.nl/uuid:ad279f6c-fbc6-4104-90b7-0a5b9e1f0088

[28] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know You Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021), 76119–76134. doi:10.1109/ACCESS.2021.3082424

[29] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 120–124. doi:10.1145/3510454.3516846

[30] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 385–396. doi:10.1145/2610384.2610404

[31] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. 2021. Research Progress of Flaky Tests. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 639–646. doi:10.1109/SANER50967.2021.00081

[32] Celal Ziftci and Diego Cavalcanti. 2020. De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 736–745. doi:10.1109/ICSME46990.2020.00083