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

Flaky Tests at Exact

George Vegelien

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

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

George Vegelien

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Exact
Molengraaffsingel 33
2629 JD Delft, the Netherlands
https://www.exact.com/

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

Author: George Vegelien

Abstract

In today's rapidly evolving software landscape, where continuous integration and continuous delivery are paramount, the presence of flaky tests poses a significant obstacle. These tests, exhibiting unpredictable pass/fail behavior, hinder development progress, waste valuable resources, and erode developer trust. This research delves into the root causes and mitigation strategies for flaky tests within a large-scale, database-driven industrial setting: Exact.

The increasing reliance on databases in modern software systems, including Exact's own platform, necessitates a deeper understanding of the unique challenges posed by database-dependent tests. By analyzing flaky test behavior through repeated test runs on the same code, we identified key contributors to flakiness, including resource contention, test order dependencies, 'dirty tests' that leave the system in an inconsistent state, platform-specific issues, and combinations thereof.

Based on the root causes for flakiness at Exact, we developed and evaluated three mitigation strategies and supporting tools: minimizing redundant database background tasks, explicitly disposing of test data, and disabling database dirty tests. Our study resulted in a substantial reduction in flakiness, leading to a significant increase in the release rate from Exact from 60% to 96%. We improved the chance of their CI/CD pipeline passing with no code changes from 27% to 95%.

Furthermore, this research highlights the importance of collecting and analyzing rich, granular test data to identify patterns and root causes of flakiness. Providing developers with actionable information from this analysis motivates them to address flakiness proactively. Moreover, understanding the interplay between different types of tests, such as the impact of dirty tests on other seemingly unrelated tests or in combination with other factors, is crucial for effectively mitigating cascading failures.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Carolin Brandt, Faculty EEMCS, TU Delft
Company supervisor:	Dr. ir. Bas Graaf, Exact
Committee Member:	Dr. A. Katsifodimos, Faculty EEMCS, TU Delft

Preface

This thesis investigates the various sources of test flakiness and explores potential mitigation strategies within the context of Exact Online, a large-scale, database-intensive industrial software system employed by a leading multinational Dutch company specializing in global business solutions. This research, undertaken as part of the Master of Science in Computer Science program at Delft University of Technology, aims to contribute to the broader understanding and resolution of test flakiness challenges faced by both researchers and industry practitioners.

I express my sincere gratitude to Prof. Arie van Deursen for his invaluable guidance and support in initiating this project and facilitating the collaboration with Exact and Caro. I am deeply indebted to the entire Exact team for their unwavering support and collaboration throughout this research. Their active engagement with my findings and their willingness to integrate these findings into their development processes have been truly inspiring. I offer a special thanks to Dr. Bas Graaf, whose mentorship and guidance have been invaluable. His willingness to discuss not only research-related topics but also broader career and professional development issues has been instrumental in my growth. I am honored to have been included in numerous company meetings and discussions, and I deeply appreciate his commitment to weekly one-on-one meetings.

I am also deeply grateful to Dr. Carolin Brandt for her invaluable support and guidance throughout this research. She not only provided detailed explanations of various topics, fostering my growth, but also demonstrated genuine care for my well-being by addressing non-technical challenges. Her insightful feedback, constructive criticism, and unwavering encouragement have been instrumental in shaping this thesis. I will miss our engaging and insightful weekly discussions, and I am truly grateful for her continuous support. Furthermore, I would like to thank Dr. Asterios Katsifodimos for providing this opportunity and assisting me in the final stages of my graduation.

Finally, I would like to express my heartfelt gratitude to my family and friends for their unwavering support throughout my academic journey. I would like to offer a special mention to my mother and sister, whose encouragement inspired me to pursue my studies in Delft. This journey has not only provided me with valuable academic knowledge but has also introduced me to a community of kind and supportive individuals whom I cherish deeply. I would like to extend my sincere thanks to these friends for their continuous support, encouragement, and the many fun experiences we have shared.

This thesis represents the culmination of a challenging yet rewarding journey, and I am immensely proud of the knowledge gained and the accomplishments achieved during this time. I am deeply grateful for the opportunity to collaborate with Exact and TU Delft, and for the unwavering support of my mentors, colleagues, friends, and family. I am excited to embark on the next chapter of my career, armed with the insights and experiences gained over the last few years.

George Vegelien
Rotterdam, the Netherlands
January, 2025

Contents

Chapter 1

Introduction

Flaky tests are the potholes in the road of progress facilitated by Continuous integration (CI) / Continuous Deployment (CD).

Similar to a pothole, a single flaky test might not be detrimental, but insufficient maintenance can exacerbate the issue, leading to cascading problems and interrupted traffic- or release-flows. Although both flaky tests and potholes are undesirable, a simple patch is not always a sufficient solution. Both can stem from easy mistakes or systemic issues such as a brittle section, a load too large, a connection point, or a combination thereof. For both potholes and flaky tests, their impact, manifestation, and root causes can vary significantly depending on the specific environment and infrastructure.

An experience probably familiar to many: you try to perform a task on your computer, but it fails unexpectedly. You then ask for help and when showing the problem by doing the *exact* same thing, it succeeds without issue. This frustrating phenomenon aptly illustrates the nature of flaky tests, where the results differ without any change. Contrary to the popular quote: '*the definition of insanity is doing the same thing over and over and expecting different results*' – it is not insanity in the realm of software testing to rerun test and expect different results, it is simply the reality of test flakiness.

The unpredictability that comes with test flakiness introduces a daily dilemma for developers. Imagine a pilot receiving a warning light on their aircraft dashboard before takeoff. They face a difficult decision: ignore the warning and risk a potential safety hazard, spend valuable time investigating a potentially non-existent issue, or repeatedly restart the aircraft systems hoping the warning disappears. This mirrors the situation developers face with flaky tests. While the safest course of action might be to investigate each warning, frequent false alarms investigations introduce wasted effort and can lead to a sense of 'crying wolf', making developers more likely to ignore future warnings, potentially masking critical issues.

This thesis delves into the world of **flaky tests**, focusing on those that occur **within complex industrial software systems that heavily rely on databases** (DBs). We explore their impact and develop several strategies to mitigate them. By understanding the root causes of flakiness and developing solutions, we can improve software quality, enhance developer productivity, and ultimately deliver more reliable and robust software applications for everyone.

Flaky tests pose significant financial and mental burdens

Flaky tests, are tests for which the outcome can either pass or fail regardless of any actual code change. They waste computational resources [71, 72], prevent automated repairs [67], burden the developers with issues that are difficult to reproduce [55, 57], create distrust in test outcomes [23] and ultimately cause developers to ignore the test results [7, 101] resulting in more crashes [85]. "*Flaky tests are one of the most significant problems for industrial software testing*" [3, 39], with over 55% of developers encountering them monthly [79]. Flaky test issues are prevalent within the entire software community, burdening students [84, 91, 96], open-source projects [54, 56, 64] and large enterprises [53, 55, 65, 71, 78], as well as spanning various software languages [36, 72, 79] and software disciplines from android development [93, 101, 2] to quantum programs [118].

The necessity for continuous research into flaky tests becomes further evident when considering its history. Practitioners recognized the dangers of flaky tests as early as 2008 (e.g., through Google blog posts [35]), with researchers addressing non-deterministic test issues long before [8, 24]. In 2011, software development pundit Martin Fowler, renowned for the *Agile Manifesto* and his book on refactoring, aptly compared flaky tests to a 'virulent infection' [28]. The usage of the Google term '*flaky test*'[1] gained significant traction that same year and has continued to popularize since [102]. This coincides with the growth of the software industry [20, 97] and the increasing number of GitHub repositories employing CI/CD technologies [19].

However, despite growing efforts from both practitioners and researchers alike [81, 99, 108, 120], the prevalence of flaky tests remains a persistent challenge. This is partly attributed to the inherent complexity of tests in large software systems, where factors impacting flakiness, such as test order dependency, have been proven to be computationally intractable (NP-complete) [119]. Furthermore, the diverse nature of flakiness [23, 64, 101] further complicate the development of applicable solutions, as some forms of flakiness arise from factors beyond our control [30]. Additionally, the inherent difficulty in reproducing flaky test failures [55, 57] reduces their priority compared to consistently failing tests [23].

To mitigate the impact of flaky tests, many developers and workflows employ the strategy of rerunning failed tests. However, this approach comes with significant costs. It not only masks underlying issues, making them harder to debug [107], but also consumes valuable computational resources. For instance, Google reported spending between 2 and 16% of its testing resources on rerunning flaky tests [71, 72]. Furthermore, allowing reruns of flaky tests until they pass in CI/CD pipelines exacerbates this issue by enabling them to persist within the system. As a result, Vassallo et al. [107] have labeled the retry strategy a 'CD smell', but it is still a common mitigation method.

Dirty tests are a root cause of flaky tests

One significant challenge in addressing test flakiness is that where flakiness exhibits may not be where the flakiness issue originates. This behavior is often associated with test order dependency, a complex issue in itself.

[1] Also referred to as 'flakey test' or 'flaky tests'.

Test-order dependency is among the top four most common causes of flakiness [23, 64, 36]. Determining whether a test is dependent on the order of execution within a test suite is an NP-complete problem [119]. A common example of order-dependent flakiness involves tests that modify a persistent state, such as database records. Consider a test that changes a database setting, causing an API to return data in an older format. Another test might assume the newer API format, leading to failure if executed after the first test. We refer to these tests that modify persistent data or leave any other type intractable 'dirt' as ***dirty tests***.

Dirty tests, also known as *polluting tests*, have been the subject of some research. However, existing research has primarily focused on identifying dependencies between tests [46, 56, 80, 110] largely through heap pollution [10, 29, 37, 119], but often neglects database-related issues. This is surprising considering that Lam et al. [55] identified this as one of the root causes of flakiness in industry. Note that dirty tests are not only test-order-dependent but can also exhibit flakiness through timing issues while the order is kept consistent, called non-deterministic order-dependent tests [58].

Exact is a large database-driven software system

Exact[2] is a multinational software company with over 2,000 employees, empowering more than 650,000 customers with its business solutions. Their flagship product, Exact Online – which for simplicity of this thesis we equate to Exact the company – is a cloud-based software-as-a-service (SaaS) business application offering a comprehensive suite of functionalities, including accounting, customer relationship management (CRM), payroll, human resource management (HRM), and enterprise resource planning (ERP). 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. Notably, the same database structure is utilized by Exact's API, Integration, and UI tests, albeit scaled down to 12 databases with mock data.

Exact leverages a CI pipeline that executes a test suite of approximately 25,000 API and integration tests roughly 100 times a day across AWS and its on-premise servers. These tests are primarily written in VB or C#, utilizing MSTest frameworks. Additionally, Exact incorporates Gherkin and SpecFlow for Behavior-Driven Development (BDD) testing in specific scenarios. One of their flaky test mitigation strategies involves a rerun mechanism, classifying a test as failed only after three consecutive failed attempts.

To deploy a new release, the main branch must successfully pass all tests, including API, Integration, and UI tests. Exact strives for daily deployments, with exceptions for specific days. However, prior to this research, their successful release rate, typically around 80%, dropped to 60%, primarily due to flaky tests and developer mistakes. Exact only releases builds that pass, they therefore requires to have at least one out of five daily pipelines running on the master branch, with all new changes, to succeed. Unfortunately, at Exact, these pipelines often failed due to **flakiness**, resulting in missed planned deployments. This significant hinders their business, as it **impedes their ability to deploy bug fixes** for customer-impacting incidents **or time-critical functionalities** such as adjustments for regulatory changes.

[2] `https://www.exact.com/`

Current database-related flakiness solutions are unfeasible

Existing literature reveals a gap between practicality and addressing flakiness for systems heavily reliant on databases. Until 2014, test-order independence was often assumed, leading to order-dependent bugs when combined with static variables or databases [119]. While research on test-order flakiness has grown significantly [99], test-order dependency through databases remains under evaluated. This is surprising given the substantial research focus on database usage in integration tests [1, 26, 114], including data population techniques [15, 25] and data flow analysis [50]. However, the connection between these approaches and flakiness is often missing.

Existing data pollution mitigation strategies, including managing test-specific data [21, 34] and techniques like in-memory databases or mocks [34], are often impractical for systems with complex databases like Exact Online's (as noted by their developers and aligned with common perceptions [95]). Similarly, restarting the test environment, while effective against flakiness, is deemed too slow by Exact's developers.

Case study at Exact of flakiness in database-reliant systems

Exact suffers from flaky tests, and developers at the company report a lack of applicable solutions in the existing literature. This gap likely arises because most flaky test research concentrates on open-source projects[81, 99, 108, 120]. While some resources exist from Big Tech companies in the form of blog posts or papers [53, 55, 65, 72, 121], there is limited research that specifically targets flaky tests in a industrial database-heavy setting.

This thesis, addresses this research gap by investigating and addressing flakiness at Exact; a large-scale, database-heavy, industrial software system. We employ a two-part approach – **characterizing the flaky tests** at Exact and **addressing the identified root causes of flakiness** at Exact – to supports our overarching goal:

> ***Goal:*** *Addressing flakiness in database-reliant industrial systems.*

Flaky data generated through same-commit test reruns facilitate fixes

To characterize flaky tests, a quantifiable gauge of test flakiness is needed. Thus, we designed a straightforward but effective approach of **repeatedly executing the entire test suite** multiple times on identical commits and aggregating their results. This enabled us **to identify flaky tests, compute metrics** such as the *pass rate* and *Flaky Pipeline Pass Percentage (FPPP)*, **and uncover patterns among flaky tests**.

We found that sharing these same-commit rerun generated reports with both developers and management fostered a significant reduction of test flakiness. Following the introduction of the initial same-commit rerun generated report, Exact achieved a consistent 3-month all-time-high 95% successful release percentage. This represents a substantial increase from their average successful release percentage of 80%, or their historically low 60% successful release percentage it had the month before.

The ***Flaky Pipeline Pass Percentage (FPPP)*** **quantifies the likelihood of a CI pipeline succeeding** when executed with identical code and data. This metric quantifies the pipeline's reliability with the rerun strategy in place. To illustrate, a 27% FPPP implies that if a developer runs the CI pipeline with no (breaking) changes, roughly three out of four times they will have misrepresenting results due to flaky failures. During this research, the FPPP proved to be a highly effective metric for **communicating the necessity** of allocating resources **to address test flakiness**. The combined FPPP of Exact's API and Integration tests rose from 27% to 95% because of the results obtained within this thesis.

The ***pass rate*** **is the average** *attempt pass percentage* for a test across multiple pipeline runs within a specific context, such as a single commit. The pass rate signifies the severity of the flakiness, separating the frequently failing flaky tests from the infrequent ones. A *pipeline run* refers to the execution of all active tests in the entire test suite, mirroring a CI step. The *attempt pass percentage* denotes the **proportion of successful test attempts** within a single pipeline run. For instance, in the case of Exact where test attempt reruns occur only upon test attempt failure: it could be that a flaky test will succeed on its first attempt in the first pipeline run and on its third attempt in the next, resulting in a pass rate of $0.67 = avg(1, \frac{1}{3})$.

Detailed explanations of these and additional metrics are provided in Appendix A.1. The data-generation method of the same-commit pipeline rerun approach and its initial results are presented in Chapter 3. The impact of sharing flaky test reports and the metrics is further explored in Section 5.1.

Our 3 research questions focus on the test environment and test dirtiness

In collaboration with the Exact developers, we analyzed the initial aggregated flaky test reports to identify the primary causes of flakiness within Exact. The initial flaky data suggested that test data availability problems or test data dirtiness were significant contributors, with the most detrimental flakiness originating from API and Integration tests. Although there were more flaky UI tests, they did not significantly impact the workflow or the FPPP, as they were well mitigated by the rerun strategy. Building on these initial results and leveraging academic research, we developed one method and two automatic tools to address the root causes of flakiness at Exact. We study the effectiveness of these approaches, we posed the following research questions:

RQ 1: *How does minimizing DB background tasks impact test flakiness at Exact?*

RQ 2: *How does explicitly disposing test-data impact test flakiness at Exact?*

RQ 3: *How does filtering database dirty tests impact test flakiness at Exact?*

Addressing these questions led to a substantial reduction in flakiness, **fixing over** 50% **of flaky tests with a pass rate of** < 0.9 **through minimizing database background tasks** and minor improvements from explicit data disposal and disabling dirty tests. A detailed explanation of the hypotheses underlying each research question is provided in Section 3.1.2, 3.1.3 and 3.1.4, followed by a detailed answer to each research question in Chapter 4.

We found that **implicit test data disposal and dirtiness are frequent violations with often, but not always, harmless effects**. We identified over 2,000 such instances among 20,000 evaluated tests, indicating a **violation rate exceeding 10%**.

We further analyzed these violating instances and their effects. Despite the high violation rate, **less than 3% of these instances induced an observable impact** on test results. However, their impact stretches far and wide throughout the code base, across non-dependent test assemblies, affecting seemingly unrelated tests.

Contrary to expectations, most of the impacting changes of **explicit disposal or disabling of dirty tests** did not fix flaky tests but instead **broke tests that relied on the specific violation or data pollution**.

These findings demonstrate that **implicit data disposal and dirty tests can lead to difficult-to-debug flakiness or brittle tests**. We provide concrete examples and detailed explanations in Chapter 4. An explanation of each automatic tool and method we employed and developed to evaluate all three research questions is given in Section 3.2 and Section 3.2.4, respectively. A quantitative analysis detailing the problem and their context within Exact are presented in Chapter 3 before these aforementioned chapters.

We show how flakiness exhibits within the industry and how to address it

Our work not only addresses the three research questions but leverages the fact that all data is obtained over time from a live and vibrant industrial system. The rampant side effects in between experiments allowed us to investigate various other influential factors to test flakiness. **We illustrate the various ways flakiness can exhibit in an industrial database-reliant system** and **the importance of information for addressing test flakiness**. Both topics have their own dedicated subsection forming the bigger part of the discussion in Chapter 5.

Our findings demonstrate that **rich and summarized information** empowers organizations to effectively combat flakiness (Section 5.1). We observed that aggregated flaky test reports not only **facilitated** the **identification of common flakiness root causes** but also effectively **communicated the criticality** of addressing these issues to developers, thereby **stimulating proactive efforts** to fix flakiness.

Furthermore, this study **demonstrates the diverse manifestations of flakiness** in an industrial setting (Section 5.2). Our findings reveal that **flakiness can arise from** a complex **interplay of factors**, leading to a variety of observed behaviors. We observed that **distinguishing between flaky, dirty, and offending tests is crucial** for effective mitigation strategies. Moreover, our analysis highlighted the concept of an 'environmentally flaky' system, where **external factors** or unstable infrastructure can **contribute to intermittent test failures**. In such scenarios, the system may eventually reach a state where ***all* tests are deemed flaky**, underscoring the importance of differentiating flakiness on its frequency.

The structure of this thesis centers around our flakiness data at Exact

The remainder of this thesis is organized as follows. Chapter 2 introduces background information, providing contextualization that aids in drawing connections to other software projects and fostering a better understanding of the research relevance, implications, and conclusions.

This chapter includes Section 2.1, which presents a partial summary of root causes, mitigation strategies, and underlying factors of flakiness as documented in existing literature. It continues with Section 2.2, which provides essential contextual information about Exact, including factors influencing flakiness, their relevant design decisions, and the way flakiness affects Exact. Chapter 3 provides a quantitative analysis detailing the problems and their context within Exact. It also describes our data-generation methodology, outlines the interconnections and significance of the three research questions (RQs), and presents the hypotheses and underlying logic supporting each RQ. Section 3.2 elucidates the tools and methods developed to investigate the RQs and details the data-generation methodology. Chapter 4 presents the results, providing answers for each RQ and a temporal analysis of our findings. Chapter 5 discusses salient findings and their primary implications. Finally, Chapter 6 concludes the research, summarizing key findings and identifying potential avenues for future work.

Chapter 2

Background

The domain of test flakiness has garnered significant attention in recent years, attracting interest from both practitioners and researchers [99]. Unsurprising given that, flaky tests are prevalent in various software systems, ranging from student assignments [91, 96] to both industrial [55, 65, 71] and open source projects [36, 54, 64]. However, despite the increasing prevalence of database-heavy software solutions [17, 32, 106], such as Software-as-a-Service (SaaS) systems [104], the study of flakiness within these systems remains relatively under-explored. We therefore investigate Exact Online, a database-reliant industrial system.

This chapter presents related work in Section 2.1 and describes the situation at Exact in Section 2.2. The related work section explores previous investigations into the root causes of flakiness (Section 2.1.1), discusses studies targeting test-order-dependent tests (Section 2.1.2), outlines other existing flakiness detection techniques (Section 2.1.3), and analyzes how researchers have designed their rerun approaches, highlighting the unique aspects of our approach (Section 2.1.4). Regarding the situation at Exact, we describe their Continuous Integration/Continuous Deployment (CI/CD) processes (Section 2.2.1) and testing infrastructure (Section 2.2.2), discuss current flaky test prevention methods employed by Exact (Section 2.2.3), explain their motivation (Section 2.2.4), and outline generalizable issues that contribute to the number of flaky tests within Exact (Section 2.2.5).

2.1 Related Work

Determining whether a test is flaky is inherently challenging; Flakiness typically manifests with unknown probabilities linked to several different, often unknown, factors. Consequently, various approaches have been designed to detect flaky tests. Some of which are outlined in Table 2.1 and further discussed in the following sections.

Section 2.1.1 begins by detailing other work that has investigated the root causes of flakiness and its impact, with a specific focus on work that investigates an industrial setting. Section 2.1.2 outlines order-dependent flakiness with a specific focus on database-related order-dependent tests. Section 2.1.3 outlines other flakiness detection techniques. Finally, Section 2.1.4 discusses which methods have relied on test reruns to gather their flaky tests and how their approach differs from our same-commit rerun method.

Study	Name	Evaluated on	Language	Goal
Lam et al. [56]	iDFlakies	Open Source	Java	Distinguish flaky tests on **order-dependency**
Wang et al. [110]	iPFlakies	Open Source	Python	Distinguish and patch flaky **order-dependent** tests
Zhang et al. [119]	DTDetector	Open Source	Java	Find heap and file **order-dependent** tests
Gyori et al. [37]	PolDet	Open Source	Java	Find heap and file **polluting** tests
Bell et al. [10]	ElectricTest	Open Source	Java	Find heap, network and file **inter-test-dependencies**
Alessio et al. [29]	PraDeT	Open source	Java	Find heap dependencies within **order-dependent** test flakiness
Huo et al. [46]	OraclePolish	Open source	Java	Find **in-memory** data that is unused or leads to **brittle** assertions
Parry et al. [80]	FITTER	x	Python	Find potential **test-order** dependencies by generating dirty tests
Dong et al. [22]	FlakeScanner	Mix	Java	Find **flaky** android GUI tests through **event-order** exploration
Gyori et al. [38]	NonDex	Open Source	Java	Find **non-deterministic specification**-reliant tests
Ziftci et al. [121]	Flakiness Debugger	Industry (Google)	Java & C++	Help developers **debug flaky** tests with execution-trace differences
Bell et al. [11]	DeFlaker	Open Source	Java	**Establish whether** test failures are **flaky** based on changes
Bell et al. [4]	FlakeFlagger	Open Source	Java	Automatically **classify** tests as **flaky** based on features
Pinto et al. [83]	ML Vocab Trainer*	Open Source	Java	Automatically **classify** tests as **flaky** based on vocabulary
Lampel et al. [60]	Telemetry Analysis*	Industry (Mozilla)	Any	**Distinguish** flaky, infrastructure, and regression-defect **failures**
O'Callahan [76]	Chaos Mode	Industry (Mozilla)	Any	Find **environmental**-susceptible **flaky** tests
Silva et al. [94]	RAFTs*	Open Source	Any	Find **resource**-affected **flaky** tests
Terragni et al. [100]	Fuzzing Test Containers*	x	Theoretical	Find **environmental**-susceptible **flaky** tests
Augusto et al. [100]	FlakyLoc	Open Source	Any	Show insights of **flakiness** inducing **environmental configurations**
Parry et al. [82]	CANNIER	Open Source	Python	Leverage ML to **preselect flaky** and brittle tests
Jiang et al. [48]	CAM	Industry (Huawai)	Any	**Suggest** high-level test **failure causes**
Lam et al. [55]	RootFinder	Industry (Microsoft)	Managed code	**Suggest root causes** for flakiness
Ours	Same-commit CI Benchmark	Industry (Exact)	Any	Evaluate a system's flaky state and **establish failure patterns**
Ours	Minimize Background Tasks	Industry (Exact)	Theoretical	Reduce task-**interference**-induced **flakiness**
Ours	WeDipose	Industry (Exact)	VB and C#	Reduce Implicit Dispose **smell-induced flakiness**
Ours	Database Sanity check	Industry (Exact)	Any	Discover database-**polluting tests**

Table 2.1: Related flakiness methods and tools. 'Language' refers to the language the method or tool is implemented for. '*' signals names we have given to methods/tools. There are many more tools that we do not cover, some of which are summarized by Zheng et al. [120] or Verdecchia et al. [108].

2.1.1 Flakiness Root Causes Investigations

Research into the root causes of flakiness and their impact has primarily focused on open source projects, particularly those written in Java and, to a lesser extent, Python. This aligns with the focus of many tools highlighted in Table 2.1, which are often developed and implemented for these languages. While some research has investigated the root causes of flakiness within industrial systems, a substantial portion of these findings are communicated through grey literature (blog posts and tech conferences) rather than academic publications. Furthermore, limited research has specifically considered the impact of databases within these flaky test root cause findings.

This section outlines other work that has investigated the root causes of flakiness. We first illustrate some of the key findings in open source work and afterwords we focus on the findings of the limited work that has evaluated industrial examples. We demonstrate that while some similar case studies have been conducted regarding flakiness within the industry, they do not provide a complete picture of the distribution of flakiness within database-heavy industrial systems.

Flakiness Open Source Studies

One of the earlier academic works investigating **flaky test root causes** is that of Luo et al. [64], who analyzed 201 commits likely linked to fixed flaky tests in over 51 open source projects, primarily Java-based. Similar to our work, their analysis included projects using .Net languages and encompassed various application domains, including web servers and database applications. They identified 10 categories of flakiness, with *Async Wait*, *Concurrency*, *Test Order Dependency* and *Resource Leak* being the most common, accounting for 84% of categorized root causes. Notably, they found that more than half of *Resource Leak* flakiness manifested within external dependencies, such as databases. Similar to our findings, they observed that some tests exhibited multiple categories of flakiness and that test behavior could be influenced by the platform on which they were executed, with 4% of categorized flaky tests being platform-dependent.

However, Luo et al.'s work differs from ours given that they did not investigate automatic mitigation approaches, and relied on bug reports linked to specific flaky test fixes. The later limits the ability to assess the overall prevalence and impact of flakiness within the system and therefore do we utilize a same commit rerun approach.

Labuschagne et al. [54] investigated the **impact of flakiness on CI failures**, finding that 13% of CI failures were attributable to flaky tests. They determined the presence of flakiness by rerunning the CI pipeline three times on the commit before, after and of the failing build. While this approach can indicate potential flakiness, it does not definitively identify all instances and may misclassify some failures. This is supported by our findings that some changes might impact the exhibiting rate of flakiness (Section 5.2.1), implying that changes that affect flakiness exhibition rate but do not cause or solve it might be misclassified as non-flaky failures with their setup.

However, Labuschagne et al.'s study was limited to open source projects written in Java. Our work expands upon these findings by investigating flakiness within a large-scale, database-heavy industrial system utilizing a different set of software languages. Moreover, instead of considering flakiness as a single entity, our focus lies on investigating and addressing the root causes of flakiness.

Gruber et al. [36] investigated the **root causes of flakiness** in 22,352 open source **Python** projects. They found order dependency to be a much more dominant problem in Python open source projects. Specifically, 7,571 (0.86%) of tests were flaky when rerunning tests 400 times, with 200 of these tests randomly ordered at the project level. Of this flakiness, 59% was caused by order dependencies and 28% by infrastructure issues. While databases were considered within their study, they were less prominent, with only 2.1% of projects being tagged as database related from the 67% of projects that were tagged by their developers.

Our work shares some similarities with Gruber et al.'s study. Both studies investigate flakiness in a less commonly studied language context (Python for Gruber et al., and a .NET environment in our case) and employe extensive test reruns with varying test-orders to identify and characterize flaky tests. Both studies also recognize the significant impact of infrastructure and environmental factors on test flakiness. However, our work differs in several aspects. We focus on an industrial setting with a complex, database-driven system, where we develop and evaluate targeted mitigation strategies to address the identified root causes. Furthermore, we investigat the impact of specific environmental factors, such as varying test platforms and scheduled background tasks, on test flakiness.

Silva et al. [94] also investigated the impact of **environmental factors**, finding that 46.5% of the flaky tests they detected were resource-dependent, exhibiting different behavior across different cloud execution platform configurations. We extend these findings by observing that platform changes often interact with other factors, such as order dependencies and test data management, to influence flakiness in complex ways.

However, our work differs significantly in the scale and complexity of the evaluated systems. Silva et al. investigated open source systems with the median test suite execution time taking 52 seconds. In contrast, our investigation focused on a large-scale industrial setting where a single test suite execution can exceed 5 hours in compute time, with certain individual tests requiring on average over 100 seconds to complete.

Flakiness Industry Scientific Studies

Several prior studies have evaluated flaky tests in large-scale industrial settings, primarily within companies with significant market capitalization (estimated to be above one billion US dollars at the beginning of 2025). These studies demonstrate that flaky tests pose a significant problem in such environments. We further found that a lot of the information regarding how industrial systems tackle flakiness is communicated through grey literature which we will discuss in the next subsection. This subsection examines how these studies evaluated flaky tests and highlights the limited information available regarding database-influenced flakiness.

Mozilla: Eck et al. [23] extended the aforementioned work of Luo et al. [64] by investigating developers' perceptions of flaky tests and their root causes. They surveyed 21 professionals at Mozilla to **classify 200 flaky tests** within the Mozilla system. They identified four additional categories of flakiness: *Too Restrictive Range*, *Test Case Timeout*, *Platform Dependency* and *Test Suite Timeout*. Their investigation revealed that 869 flaky tests were fixed within a year, indicating a significant prevalence of flakiness within an industrial setting, with over two flaky tests requiring attention each day.

While Eck et al.'s work provides valuable insights into developers' perspectives on flakiness, it shared some limitations with Luo et al.'s study. Unlike our work, neither study investigated the distribution or impact of flaky tests within the system.

Mozilla has been a subject of other research investigating flakiness in industrial settings. Rahman et al. [85] investigated **the relationship between flakiness and crash reports**, while Lampel et al. [60] investigated **the percentage of CI failures due to flakiness** and the ability to detect these root causes. In Mozilla, dedicated engineers called 'sheriffs' classify CI failures as either system bugs or flakiness [60]. Lampel et al. found that, within their examined subset of test suites, 25,871 (67.0%) of the 38,596 (2.1% of all pipelines) pipeline failures observed over four months were caused by flakiness.

While these three studies provide valuable insights into flakiness within Mozilla, they also have limitations. Eck et al.'s and Rahman et al.'s studies focused on known flaky tests, potentially introducing a bias towards more easily detectable or impactful instances. Lampel et al.'s investigation, while examining all temporally obtained flaky test results, might be biased towards areas with high code change throughput as their CI utilizes test selection, potentially overrepresenting flakiness in certain areas of the codebase. Moreover, the impact of database interactions on flakiness was not explicitly considered in any of these Mozilla-specific studies. None of these studies explicitly address the concept of brittle tests or the intricate relationships between multiple factors that can influence test flakiness.

Ericsson: Malm et al. [86] and Rehman et al. [66] both investigated approaches to addressing test flakiness within Ericsson's software system. Malm et al. investigated **the effects and distribution of delays**, with the goal of creating a static analysis approach to detect potential flakiness related to the usage of sleep-type delays, a type of flakiness that was not the primary focus of our investigation. Rehman et al. measured **the number of No-Fault-Found (NFF) test failures** (i.e., test failures with no subsequent bug report) and investigated the effect of communicating this measurement. They defined two new metrics: *NFFRate* and *StableNFFRate*, which quantify the percentage of test failures that did not result in a bug report compared to the total number of runs. The 'Stable' prefix refers to whether the system is stable. The *StableNFFRate* is essentially the same as our *pass rate*, given that we always rerun on a stable release, with the difference being that we account for attempt reruns and our rates are flipped (i.e., a StableNFFRate of 0 is equal to a pass rate of 1). They observed that 18% to 22% of tests had more than 10 NFF failures, which is substantially higher compared to our equivalent (tests with a pass rate less than 0.9), which was less than 0.7% of tests. Similar to our work, they found that providing the NFFRate to developers helped them quantify the problem instead of relying on intuition. However, their work did not delve into identifying and addressing the root causes of flakiness as extensively as our study. Lastly, their work performs statistical tests on the pass rate under the assumption of a binomial test distribution. However, we did not make this assumption, as we do not consider test failures to be independent observations.

Apple: Kowalczyk et al. [53] investigated **the effects of temporal flakiness scoring** at Apple. Similar to the work of Rehman et al. [66], Kowalczyk et al. designed a scoring system to identify, quantify, and rank test flakiness. They formalized two models: one based on *entropy* (a metric from information theory quantifying the disorder or uncertainty of a random variable) and another called *flipRate*, which measures the rate at which test results transition between passing

and failing. These metrics were then adapted for a stronger temporal relationship by integrating version history, similar to a weighted moving average in a time series approach. The authors used these models to observe flakiness trends, reduce flakiness, and identify environmental and test suite causes. Their findings indicated that quantifying flakiness with their scoring system enabled them to identify flaky causes and subsequently reduce flakiness by 44%. Consistent with our findings, they concluded that "simply identifying flaky tests is not enough; quantifying test flakiness enables us to better understand test behavior." However, our work differs in that we utilize our *pass rate* not only for fault localization but also to evaluate the impact of our mitigation strategies. Furthermore, within our study we explicitly address the root causes of flakiness and develop and evaluate mitigation strategies, which their work did not.

Microsoft: Herzig et al. [43] investigated whether they could detect 'false test alarms' (which we classify as flaky tests) by utilizing **association rule learning to identify patterns among failing tests to automatically classify tests as false alarms**. They found that less than 5% of test failures were due to false alarms. They were able to detect flakiness with precision between 0.85 and 0.90, detecting between 34% and 48% of false test alarms, resulting in an estimated saving of 100 minutes per day of blocked Windows CI steps due to flakiness. While we also investigate and find flakiness based on common patterns, we focus on analyzing error messages instead of test case names and test steps.

Flakiness Industry Grey Literature

Most companies that highlighted the issues with test flakiness communicated their findings and beliefs through grey literature, such as articles, blog posts and tech talks.

Martin Fowler, in collaboration with Thoughtworks [28], discussed observed **root causes of flakiness**, including insufficient test isolation and resource leaks. P. Sudarshan, from the 'Go team' at Thoughtworks [98], further elaborated on these findings, detailing their five-step workflow – stop accepting, quarantine, plan, refactor, and learn – which they utilized to address flakiness. N. Mellifera, from Signadot, observed that the **scale of the system and the number of teams were related to the prevalence of flaky tests** [68]. Numerous other articles on Medium (over 90 tagged with 'Flaky Tests' [69]) offer insights into industry experiences with flakiness. However, most of these articles are shorter pieces aimed at informing, motivating, or highlighting potential mitigation techniques, lacking the depth and rigor of more formal research.

Google: More strongly funded companies often support their blog posts or tech talks with data-driven empirical backing. Two such articles, written by J. Micco [70] and J. Listfield [63] at Google, have become points of reference for engineers at Exact. Micco's article **listed high-level flaky test mitigation strategies at Google**, while Listfield's examined **the correlation between test size and flakiness**. These articles provide a high-level overview, but do not delve into the complexities of flakiness with the same level of detail as our research.

Through tech talks, Micco et al. [72] reported that 84% of test failures at Google were considered flaky and that 16% of their tests experienced some level of flakiness. To mitigate this, they employed a **flakiness score based on the variation of a test's results over time**. They also found that failures associated with configuration files were highly likely to be flaky. These findings align with our observations that most tests at Exact will eventually be deemed flaky and that changes in the start configuration, such as minimizing database background tasks,

significantly impact test flakiness. However, our research more deeply investigates the root causes and patterns of flakiness, incorporating a more comprehensive analysis of test behavior and environmental factors.

Spotify: J. Palmer **investigated and addressed test flakiness at Spotify**, showing the prevalence of order-dependent tests caused by a global state [78]. He highlighted the necessity of a flakiness score and observed that providing developers with a view of flaky tests reduced flakiness at Spotify from 6% to 4%. While these takeaways are similar to our findings, our study has a stronger focus on detailed root causes and developes targeted mitigation strategies.

Facebook: Machalica et al. [65] investigated **the effects of a probabilistic flakiness score** (PFS) metric at Facebook, utilizing techniques such as entropy and flip rates to assess test stability over time. They found that all tests exhibit some degree of flakiness and that a gradient scale is necessary to effectively quantify this variability. Similar to our study, they quantify and address test flakiness. However, our work focuses on investigating the root causes of flakiness and quantifying test or test suite flakiness across different versions of the software, while Machalica et al. primarily focused on the construction and impact of a *temporal* metric (PFS) for individual test flakiness.

Flakiness Industry Case Studies

Other work has attempted to reduce the number of flaky tests in an industrial setting by studying tools that provide developers with additional information. One study was conducted by Ziftci et al. at Google [121] and the other one by Lam et al. at Microsoft [55].

Google: Ziftci et al. developed an algorithm called *Divergence* to **compare execution traces between passing and failing runs** [121]. They integrated *Divergence* with the *flakiness score* [72] within their tool, Flakiness Debugger (FD). FD instruments all tests with a specific flakiness score to track and calculate common divergence points for flaky tests, enabling it to identify and notify developers of common and divergent execution traces for detected flaky tests. They evaluated the usability of this tool through various approaches, one of which exhibits similarities to our own. Specifically, they analyzed identified flaky tests over a defined period, provided this feedback to developers, investigated the number of flaky tests resolved at the end of the period, and queried developers regarding the helpfulness of the additional flakiness information. Similar to our work, they generally target any type of flaky test observed in practice within the system, including order-dependent tests. The only filtering applied to flakiness is analogous to our distinction between sporadic and non-sporadic flaky tests; They evaluated their tool exclusively against tests with a high *flakiness score*, i.e., tests that often exhibit flakiness. Their findings indicated that 166 out of 300 FD-applied existing flaky tests were resolved by the end of the study.

A key difference between our work and that of Ziftci et al. lies in their lack of investigation into the flaky test distribution within the system. They evaluate flaky tests that are already deemed flaky based on their own time-series metric, the *flakiness score* [72]. Secondly, while this flakiness score incorporates order-dependent tests, their tool (FD) is not actually evaluated with a test independence consideration, as FD reruns tests in isolation. Finally, they do not specifically mention the integration of databases within their tests.

Microsoft: The work of Lam et al. [55] most closely resembles our own, as it **investigates the distribution of flakiness within an industrial database system**. Lam et al. investigated the root causes of flaky tests in a large-scale industrial setting and developed a tool (**RootFinder**) to assist developers in identifying these root causes. They examined several projects at Microsoft by monitoring their CI pipelines and identifying flaky unit tests through automatic reruns whenever a test fails. They subsequently reran all flaky tests 100 times with instrumented .NET dependencies to generate log files containing various runtime properties during different execution points, for both passing and failing attempts. For reproducible flaky tests, they further investigated root causes and determined how developers can be aided in this task by providing more information through their tool.

This study shares numerous similarities with our work and is one of the few studies that investigate the root causes of flakiness within industrial and database-reliant systems. It explores how information obtained through test reruns can assist developers in debugging flakiness and also shares many similarities in terms of technical details, focusing on .NET applications and leveraging MSTest for managed code (e.g., C#).

However, there is one fundamental difference: While they focus on individual flaky tests, we investigate flakiness as a whole. Their root cause analysis method relies on the assumption of test independence, an assumption that we found to be untrue for most root causes of flakiness at Exact. Although they mention that RootFinder can detect test-order-dependent flakiness, they indirectly filtered such tests in their evaluation. This is because they employed a fixed test-order and only evaluated tests where they could reproduce flakiness by rerunning *only* the flaky tests instead of the entire test environment. This approach, along with various other confounding factors such as platform changes, resulted in flakiness reproducibility for only 44 out of 315 tests. Our work focuses on these confounding factors and inter-test dependencies by leveraging whole-environment reruns to investigate flakiness within the entire system, including background task configurations and resources, to determine underlying root causes.

2.1.2 Test Dependence Detection Techniques

Prior research have investigated inter-test dependencies, particularly focusing on test-order-dependent flakiness. While some work does not consider order-dependent test as flaky These studies typically employ strategies such as varying test-orders or investigating data connections between tests. However, many of these approaches do not adequately consider the influence of databases or other confounding factors. This section provides an overview of prominent order-dependent tests and relevant research that incorporates dirtiness.

Zhang et al. [119] were among the first to address the test independence problem. They **investigated order-dependent tests and their corresponding fixes** by examining software issue tracking systems. Subsequently, they designed and evaluated **DTDetector**, a tool equipped with four algorithms: two targeting general order-dependent tests, one focusing on k-wise order-dependencies (i.e., test-order dependencies between k tests), and another targeting either brittle tests (with $k = 1$) or all k-wise in-memory order-dependencies (if $k > 1$). While our research indirectly leverages one of these algorithms, as Exact relies on randomized test-orders, our focus extends beyond order-dependencies to encompass a broader spectrum of flakiness types and their combinations. For example, we investigate the impact of test-order-dependent tests exhibiting non-deterministic behavior with fixed test-orders, such as those associated with the WeDispose smell.

Hou et al. developed **OraclePolish**, a tool that utilizes a dynamic tainting approach to **identify all brittle tests**, pinpoint the specific assertion responsible for the brittleness, and identify all unused inputs [46]. The tool's ability to detect brittle tests aligns with the objective of our database sanity check: to identify all potential data leaks. However, their work does not incorporate data flow within databases and, in its current state, does not account for shared data, a prevalent characteristic in most API and Integration tests at Exact. These tests at Exact typically require pre-populated databases for faster and more comprehensible execution. Furthermore, OraclePolish does not consider dirty data that can lead to exceptions prior to the assertion. We believe that taint analysis holds significant potential as an alternative to sanity checks and further elaborate on its possibilities in Section 6.2.

Lam et al. developed three frameworks: **iDFlakies** [56], **iFixFlakies** [92], and **iPFlakies** [110] to **detect, classify, or fix order-dependent flaky tests**. iDFlakies, the oldest framework, identifies order-dependent flaky tests within git repositories. It classifies tests as either flaky or non-flaky and determines order-dependency by rerunning the entire test suite on the same commit using four distinct approaches that systematically alter the test-order. Similar to their 'Running' step in the random-class approach, where they vary the order of test classes but not methods, we semi-randomly[1] vary the order of test assemblies within our same-commit rerun approach.

One key difference between our approach and iDFlakies lies in their exclusion of all Maven modules containing a failing test in their original order. This exclusion filters out always first-attempt-failing tests, a type of flaky behavior frequent and impactful at Exact.

iFixFlakies [92] is designed to automatically patch a given order-dependent test when provided with a failing and passing order for that test, which can be identified by iDFlakies. iPFlakies [110] encapsulates the functionalities of both iDFlakies and iFixFlakies and make it applicable for Python code. All three frameworks differ from our work in that they are evaluated for different programming languages and do not investigate an industrial system or specific types of order dependencies.

Gyori et al. designed **PolDet** [37], a technique to **identify tests that pollute a shared state** by capturing the state before and after test execution. PolDet can recognize changes in heap and file systems but not databases or any network-connected storage systems. Our work aims

[1] We consider our test-order random, but in reality certain order constraints exist, such as the order of test execution.

to address this limitation by developing the database sanity check. Our database sanity check approach operates similarly to PolDet, instrumenting the test framework's test initialization and teardown methods to compare the test data pre- and post-states. The big difference between our work is that instead of focusing on heap and file system dirtiness, we target database-dirtiness. Secondly, we evaluate our work in terms of its impact on flakiness and assess it within an industrial system, unlike previous research which primarily focused on open source systems.

Bell et al. developed two approaches: **ElectricTest** [10] and **PraDeT** [29] to **identify data dependencies between tests**. ElectricTest focuses on identifying any data dependencies between tests, including cases where a test reads data which is or will be written to by another test, or where multiple tests utilize a side-effect-free method that is cached by the system after the first test uses it. ElectricTest achieves this through instrumentation of all classes within the system under test (including its libraries), enabling it to monitor all global resource reads and writes, as well as the socket addresses of used external data for each test.

It is important to note that ElectricTest only identifies data dependencies. While these dependencies can potentially lead to flaky tests, the research by Bell et al. does not directly target flaky tests and, therefore, does not evaluate ElectricTest's ability to identify flaky tests.

In contrast, PraDeT specifically targets flaky tests. PraDeT combines the precision of DTDetector with the speed of ElectricTest to **identify all global data dependencies** for all order-dependent tests. It employs an iterative approach, testing all global data dependency chains and filtering for only 'manifest' dependencies (dependencies with an observable impact). Subsequently, it tests all chains of manifest dependencies using an varying test-order approach similar to that of DTDetector or iDFlakies. This enables PraDeT to identify intricate test-order-dependent tests, including those that rely on one or more other tests.

Similar to our same-commit rerun approach with varying test-orders, PraDeT aims to identify flakiness caused by order-dependent tests within a system, including longer chains of order-dependent tests. While their work automates the narrowing down of these relations and conducts a more comprehensive search, it does not account for external files or file systems, such as databases. This relationship with databases is precisely the connection we aim to investigate. Additionally, their method only considers deterministic order-dependent tests. Instances that exhibit non-deterministic behavior within a given order, such as the Implicit Dispose smell, will not be detected.

Parry et al. adopted a different approach than those described above. Instead of detecting test dirtiness, they leverage it. They propose **FITTER**, a tool that **proactively reveals potential test-order-dependent tests** by generating dirty tests. Similar to fuzzing for system bug detection, FITTER aims to induce test failures by intentionally introducing dirty test data to uncover potential order dependencies.

Our study shares a similarity with FITTER: both aim to demonstrate how dirty tests can cause other tests to become flaky, enabling engineers to recognize the same behavioral patterns in other scenarios. Similar to our approach, FITTER reveals various types of order-dependent flakiness, potentially including combinatorial causes of flakiness (Section 5.2.1). This occurs in FITTER because it automatically generates entire tests instead of solely manipulating data. However, we believe this approach may not be well-suited for Exact, as it often relies on data-cleaning rather than data setting. FITTER would therefore likely identify a large number of

potentially problematic test-data relations. Instead, we adopt a reactive approach, investigating test-order dependencies that have already resulted in flakiness. The distinction between relying on data-cleaning and data setting is discussed in more detail in Section 5.2.2.

Database Test Dependence

As illustrated we see that order dependency can cause test flakiness by manifesting dirt in persistent data. While these aforementioned approach work well for variable or file dependencies, they are not applicable for database dependencies. Meanwhile one of the earlier work regarding test independence, the aforementioned work by Zhang et al. [119], already warned regarding the danger of test dependence manifesting through databases. Nevertheless, research specifically focusing on database test dependence and the various different flakiness complications because of it remains limited. Although, related work has investigated the impact of factors contributing to flakiness, such as database dirtiness, concurrency problems, or test configuration, it has often failed to account for the interplay of these factors.

For instance, Lam et al. [55] acknowledged concurrency issues within databases as a potential source of flakiness, but only within the context of a single test. Similarly, Dong et al. [22] designed **FlakeScanner** to **explore all possible event orders** within Android GUI tests **to identify concurrency-related flaky tests**. Although their targeted and evaluated tests include those that connect to databases and APIs, FlakeScanner only considers concurrency-related flakiness within the scope of a single test, neglecting database-related test-order dependencies. We capture all database-related test-order flakiness by rerunning the whole CI with various test-orders.

Augusto et al. [73] focused on web applications with databases. They developed **FlakyLoc** to **identify flaky tests in web-based testing by varying environmental factors** such as network speed, CPU usage, browser type, screen resolution, and operating system. Whereas our research targets similar systems and addresses flakiness through configuration redesigns, we do not investigate UI tests or redesign different configurations in the same manner, only background task configurations. Importantly, while their work incorporates databases, it does not explicitly address database-specific issues or consider the impact of database test dirtiness within the system.

2.1.3 Flaky Tests Detection Techniques

Several tools proactively identify potential flaky tests by focusing on behaviors within tests that can lead to flakiness. These approaches effectively reduce the number of flaky tests within a system. However, they differ from our approach as they do not index *all* types of flaky tests that are *currently* exhibiting in Exact. One such example is NonDex.

Gyori et al. presented **NonDex** to **detect and debug wrong assumptions on Java APIs** by randomly exploring different behaviors of underdetermined APIs during test execution [38]. NonDex proactively identifies potential flaky tests by detecting tests that rely on non-deterministic specifications of APIs. For example, iterating over a HashSet in Java 7 produces a different order than in Java 8, potentially breaking tests that rely on a specific iteration order (e.g., a test verifying whether the first element in the set is 42). This approach differs from ours as we investigate

all types of flaky tests that are *currently* exhibiting in Exact, including tests that exhibit flakiness without platform or configuration changes.

In contrast, Bell et al. designed **DeFlaker** to **determine whether a failed test is caused by flakiness** or a code change without the need for rerunning [11]. DeFlaker leverages statement- and class-level 'differential coverage', which compares coverage information between tests run before and after code changes, to evaluate whether a test failure is due to the code changes. This method is a proactive approach to flaky test evaluation and may falsely flag more tests as flaky than are actually problematic. Therefore, it is not intended to index the currently exhibiting flakiness within a system. Furthermore, DeFlaker currently does not support external files, such as configurations and databases, which is an essential factor we focus on.

Machine Learning Techniques

Several works have leveraged **machine learning techniques to detect and classify root causes of flaky tests**. These works, such as **FlakeFlagger** [4] and Pinto et al. [83], focus on similar types of flakiness and, like our root cause analysis, utilize related tests, test attempt information, and language-processing techniques to distinguish, group, and classify test failures. While we also consider certain test characteristics, such as execution time, our approach differs significantly. Instead of using these characteristics to predict whether a test is flaky, we employ them as clusterable factors to aid engineers in finding root causes of found flaky tests.

Other work has focused on **root cause analysis for test failures** using machine learning techniques. Lampel et al. [60] performed root cause analysis using classification models based on telemetry data of failed tests. While we also categorize failures based on telemetry data, such as execution times, we do not employ machine learning techniques for this purpose. We also primarily categorize failures based on error messages and pass rates, and utilize these categories to identify, test, and evaluate prevention approaches instead of distinguishing among flaky, infrastructure, and regression-defect test failures.

In addition, Jiang et al. developed **CAM**, which investigates failure causes of tests at Huawei-Tech Inc. based on test logs [48]. They employ natural language processing on their test logs (including test error messages) to **detect and match test failures to previous failures**. This shares similarities with our work, as both approaches involve grouping test failures in an industrial setting based on common overlapping phrases in error messages. However, our work utilizes this information to find unknown common root causes for flaky tests, while Jiang et al. use it to distinguish between test alarms. These test alarms are more abstract than our error message categories or flaky causes, and they focus on test failures for any reason (including product code defects), not specifically on flakiness. While some test alarms, such as 'C5 – Device anomaly' and 'C6 – Environment issue', relate to flakiness, we consider these to fall under external or environmental factors influencing test flakiness.

Environment Configuration Fuzzing

Several tools and methods exist to reveal flakiness influenced by environmental factors. **Chaos Mode** exercises nondeterminism in tests to simulate possible variance in **bug-inducing factors related to the test environment**, including platform, hardware, or available resources [76].

Silva et al. introduced another method for **restraining resources to uncover Resource-Affected Flaky Tests** (**RAFTs**) [94]. Terragni et al. presented a **theoretical approach to detect environmental root causes of flakiness** by fuzzing the execution environment of the test across a particular dimension through different execution clusters [100]. While our work incorporates the effects of environmental factors on flakiness, we investigate how these factors manifest in reality within Exact's CI system, rather than attempting to artificially create or recreate related test failures.

Combinations of Techniques

Some research has combined several methods mentioned above. Parry et al. introduced **CANNIER**, an approach that combines rerunning-based flaky-detection techniques with machine learning models [82]. They found that by combining their machine learning approach with rerunning approaches, such as iDFlakies, they could **preselect which tests are or are not flaky**, reducing time costs by an average of 88%. Similarly, Lam et al. also leverage iDFlakies in combination with NonDex to **investigate the diminishing returns of** found flaky tests compared to **tool execution frequency**, aiding in evaluating the benefits and total time costs [59].

While both of these studies focus on indexing a combination of several root causes of flakiness present within systems, they primarily focus on open source projects, unlike our research, which investigates an industrial context. Additionally, although their work also focuses on test independence, including database-dirtiness and brittle tests, they do not fully incorporate the possibilities of combined flakiness types. For example, we investigate the Implicit Dispose smell, which can result in dirtiness for a non-deterministic amount of time, potentially leading to order-dependent behavior with fixed test-orders.

2.1.4 Rerunning Benchmarks

Several studies have relied on test reruns to measure flakiness within a system. Some approaches extensively rerun flaky tests 10,000 times [4]. However, subtle differences exist in the implementation details of these rerun methods, leading to varying results. Notably, no other work performs reruns in the same manner as our approach, which involves rerunning the entire CI pipeline with the same repository-wide commit and utilizing varying test isolation levels between test reruns within a pipeline. This section discusses how other rerun methods differ from our approach. Many of these papers and methods are illustrated in previous sections, so this one focuses specifically on the flaky data aggregation methods employed through test rerunning.

Some work relied **solely on rerunning known flaky tests**, either from a pre-defined set or tests identified as flaky within a specific time period. This was employed by Pinto et al. [83] and Lam et al. [55]. However, this approach results in the concealment of unknown flakiness and filters out order dependencies that involve tests not deemed flaky.

Bell et al. [11] are to our knowledge the first to explicitly **evaluate different data-isolation levels between rerun approaches**, all allowing up to 5 rerun attempts. At the lowest level, they evaluated Maven's Surefire test runner rerun approach, which executes the same test within the same JVM where it initially failed. The second level involved running the test in a clean, new JVM, and the final level rebooted the machines and executed `mvn clean` between reruns. This

analysis revealed that 1,162 (22.9%), 3,024 (59.23%), and 889 (17.5%) of the 5.075 flaky tests passed with at least the respective isolation level, respectively ordered from lowest to highest. Within our work, we employed random isolation levels, either their highest or lowest level, by randomly redistributing failing tests to either the same or a different VM. They evaluated 5.966 commits from 26 open source projects once using all three respective data-isolation rerun strategies on their local hardware. Our work differs from theirs in that we reran on the same commit on the dedicated CI hardware with the failed test rerun strategy in place.

Zhang et al. [119] evaluated both already identified flaky tests and their corresponding patches, and **reran** 4 open source projects linked to a **specific commit up to 1,000 times with random test-orders**. They conducted the latter on dedicated hardware and found that 29 (0.7%) human-written and 345 (5.5%) automatically generated tests were order-dependent. Similarly, Lam et al. [56] reran the test suite of numerous open source projects linked to a specific commit 8 or more times in its entirety with varying test-orders, identifying 422 (less than 0.5%) flaky tests belonging to 82 projects, of which 213 were order-dependent.

Lam et al. [58] **reran** 26 test suite modules containing known flaky tests, from 23 open source projects linked to specific commits, 4,000 times on Microsoft Azure on **the same commit with random test-orders**, identifying 107 flaky tests. They subsequently ran these 107 tests 4,000 times in isolation and found that only 46 of these exhibited flaky behavior in isolation, while 4 were brittle. This work also addresses non-deterministic order-dependent tests, which we address in our work. However, there is a slight categorical difference, as they refer to them as a subcategory of non-order-dependent tests, whereas we consider them to be order-dependent. While this work exhibits similarities to our rerun approach, they did not rerun the project on its default CI hardware and only reran Maven modules containing known flaky tests. This limits the identification of certain intricate relations causing test failures across assemblies.

To mitigate the impact of skipping modules with an unknown number of flaky tests, Alshmmari et al. [4] reran the entire test suite of 24 projects 10,000 times. However, our approach differs from theirs in several key aspects: they did not employ a failed test rerun approach, they utilized a static test-order, and they executed the tests on their local hardware instead of the hardware used for production test deployment.

To our knowledge, we have implemented a unique rerun approach by combining several of the methods mentioned above. We have employed a same-commit, complete test suite-wide test execution, similar to Alshmmari et al. [4]. Differing from that work and similar to that of Lam et al. [56, 58, 110], we employed varying test-orders between assemblies. The aforementioned works did not utilize mixed data-isolation levels between test reruns. Similar to the work of Bell et al. [11], we employed test reruns with multiple data-isolation levels, but we did so randomly and interchangeably, allowing up to a maximum of 3 attempts per test per test suite execution. The final key design decision that distinguishes our rerun approach and impacts the identification of flaky tests is our execution of test suites on the default CI pipeline configuration and hardware. This will result in different tests exhibiting flakiness, as demonstrated by Silva et al. [94] and Terragni et al. [100] in their investigation into the impact of explicitly varying the test environment, platform, and configuration. Within our study, we also varied the test platform (Section 4.1.3) and the start configuration (Section 4.2) between test suite runs.

Furthermore, a technical distinction exists between our rerun approach and that of most

flaky test research, as our approach is specifically designed for MSTest in conjunction with VB and C#, instead of Java or Python. Previous research has demonstrated that software languages exhibit different correlations with test flakiness [72] and may exhibit a different distribution of root causes [36]. We designed our rerun approach for MSTest because it is the framework employed by Exact, as discussed further in the next section.

2.2 Situation at Exact

This section provides an overview of the relevant aspects of Exact's software development environment, including its Continuous integration (CI) / Continuous Deployment (CD) pipelines, testing infrastructure, and current approaches to mitigating flakiness.

We begin by describing Exact's CI/CD pipeline in Section 2.2.1. This includes details about the stages involved, the role of Feature Release Tests (FRTs) and Release Regression Tests (RRTs), and the release process. We then delve into the specifics of Exact's testing infrastructure in Section 2.2.2, focusing on the hardware and software used for testing, the parallelization of test execution, and the management of database instances. In Section 2.2.3, we explore the methods currently employed by Exact to prevent flakiness. These include strategies such as test retries, dedicated database instances, and the utilization of multiple instances of templated data. Section 2.2.4 describes some of exact its current motivation to fix test flakiness. Finally, in Section 2.2.5, we discuss some of the generalizable challenges faced by Exact's testing environment, such as the presence of legacy code, the contributions of a large and diverse developer community, and the inherent challenges of testing in a cloud environment with dependencies on external services.

2.2.1 The CI/CD of Exact

Exact has some noteworthy infrastructure configurations with regard to testing, Continuous Integration (CI), and Continuous Deployment (CD). These configurations are relevant to know where test flakiness can or can not occur, how it enters production code, and how it can affect deployment. The following sections explain what is needed for code to be merged, when the company releases updates, and the setup of its technical infrastructure.

Merging requires a code review and successful build and FRT

Like many other businesses[19], Exact employs a CI/CD pipeline. The CI phase of this pipeline involves building source code and executing unit, API, Integration and optionally UI tests, among other tasks. Within Exact, a CI pipeline run is referred to as a Feature Regression Test (FRT) run. Do not confuse this FRT/CI pipeline with the **pipeline** used throughout the rest of this thesis and defined in Appendix A.1, which only encapsulates the API and Integration tests stages of the CI pipeline run.

A code review and a successful FRT are required to merge code to the master branch. However, both mechanisms may not effectively prevent all types of bugs. To illustrate, when an FRT is requested during a Merge Request (MR), the new code is pulled into the master branch, and the FRT is executed on this code. When successful no additional FRTs for up to 72 hours on the

MR are need, possibly leading to new test-failing commits to be included. Furthermore, there are always several in-flight MRs and FRTs do not incorporate these concurrent changes, however, certain bugs may manifest through a combination of the changes. To mitigate these risks, Exact mandates a successful Release Regression Test (RRT) run before deploying any updates.

Exact releases daily updates based on the last successful pipeline run of the master

An RRT run is similar to an FRT run, but with key differences. Firstly, RRTs are automatically triggered several times a day and exclusively for code merged into the master branch. Secondly, they are executed on more powerful hardware than FRTs and with a larger number of test agents. If an RRT fails, developers are assigned to investigate the issue. To ensure timely resolution, RRTs are only run during working hours. This limited time window, combined with the 1.5-hour duration of each run, restricts the number of daily executions.

Exact aims to release an update daily at 4:00 a.m., before the start of the workday. This approach enables the delivery of smaller feature sets and quick shipment of bug fixes. The last successful RRT of the previous day is deployed.

This context illustrates the necessity of a high flakiness pipeline pass percentage (FPPP), which describes the likelihood of a pipeline passing without code changes. A lower FPPP increases the frequency of developer assignments to investigate test failures. These failures are challenging to identify due to the inconsistent impact of code changes on test behavior. Moreover, a lower FPPP decreases the likelihood of a successful Release Regression Test (RRT), hindering the ability to release bug fixes and new features on a daily basis. Furthermore, the longer between passing RRTs the more the code diverges between RRTs, thereby increasing the complexity of failures of the next RRT.

Integration, API and UI tests are the only stages considered to be flaky

Within the FRTs, a consistency assumption is made: All steps except for integration, API, and UI tests, are considered non-flaky. This implies that unit tests are reliable, and a single failed unit test attempt can lead to the failure of the entire FRT. Consequently, when discussing 'the pipeline' in this thesis, most CI pipeline steps, including unit tests, are disregarded. UI tests are also excluded due to their unique test environment configuration, which falls outside the scope of this research. This is further elaborated in Section 3.1.1.

2.2.2 Exact Testing Infrastructure

This subsection delves into the technical infrastructure employed by Exact for testing, providing a foundation for understanding the context of our research. We describe the hardware and software components involved in the testing process, including the use of virtual machines, dedicated hardware, and parallelization techniques. We also discuss the management of database instances, including the use of templates and the strategies employed to ensure data isolation and prevent interference between tests.

FRTs use virtual machines with dedicated hardware

The API and Integration test stages of the FRTs run on virtual machines (VMs) hosted in a dedicated data center owned, maintained, and used solely by Exact. All servers for the FRTs in this data center utilize identical physical hardware. This eliminates physical speed differences between VMs, except for potential variations in host machine workload. This approach effectively mitigates flakiness caused by hardware discrepancies ([94, 23]). The implications of this are further discussed in Section 5.3.3.

A partial cloud migration to AWS occurred during this study, taking place between the experiments for RQ2 and RQ3. Consequently, approximately 20% of all pipeline runs for RQ3 are executed on AWS hardware. All other runs, including the ones for RQ1 and RQ2, are conducted solely on Exact's hardware. The potential impact of this migration is addressed in Section 5.3.4.

Test are randomly distributed over 2-5 test agents to provide test parallelization

API and Integration test stages parallelize their execution using multiple test agents, with 2 and 5 agents for each stage, respectively. Each test agent corresponds to a single VM. When a test stage starts, each VM boots up a clean database instance from a pre-defined template and executes a subset of the tests sequentially. This ensures that tests do not compete for database resources, and only one test at a time interacts with the database.

The distribution of tests across test agents is dynamically managed by a *First In, First Out* (FIFO) queue. This real-time distribution approach offers adaptability to changing schedules and its simplicity in implementation. Furthermore, it is near-optimal in terms of absolute execution time; It can take at most the difference between the longest and shortest job, exceeding the optimal solution [33].

Tests are grouped by their test assembly and loaded into the FIFO queue in alphabetical order. A test assembly refers to a collection of test classes and their associated tests compiled into a single .dll file. The FIFO queue distributes test by its assemblies to available test agents, therefore does the test order between test assemblies vary, while the order within a test assembly remains consistent. There are roughly 300 test assemblies, containing between 1 and 900 active tests and a median of 34, for the API and Integration tests.

If a test attempt fails and has not reached its maximum allowed attempts, it is re-queued at the end. When this re-queued is executed again, it performs all assembly-, class- and test-level, initialization and cleanup operations. This re-execution might be executed within a different VM and *only* the failing tests within an assembly are executed again.

The database instances used in testing reproduce production by having over 18,000 tables but are filled with mock data

Each database instance is established from the same template to resemble the production database only scaled down to 12 databases instead of several hundreds. They are populated with mock data and mock users and use a read uncommitted isolation level and sometimes used in combination with Redis[2] to test and handle caching. A single database instance is a Microsoft SQL

[2] `https://redis.io/`

Server 2022. The test server has 12 databases each with over 1500 tables. The size of this database poses some difficulties further discussed in Section 3.2.1.

All ± 25,000 API or Integration tests are written in VB and C#

During this thesis, the number of active API and Integration tests within Exact Online ranged from 6,700 to 8,000 and 16,800 to 18,000, respectively. All tests are executed during every RRT and FRT run, with no selection based on code changes. This significant test count variation reflects the scale and continuous evolution of the system and its tests.

All tests are written in either VB or C#, and MSTest[3] is used for test execution. Approximately 19% of Integration tests and 1% of API tests are generated by SpecFlow[4] and therefore excluded from our adaptations (Section 3.1.1).

While there are no technical distinctions between API and Integration tests, API tests are not intended to directly access the database. However, in practice, such access is still possible, and sometimes utilized in the setup, verify or cleanup phase of the test.

FRT and RRT configurations are similar

This study does not extend to the configuration of RRTs, as only the FRT configuration was used for experiments. The observant reader might be curious about the discrepancy mentioned in Appendix A.2, which describes RRTs as having a different than FRTs. We consider these changes to be minimal and outside the scope of this research, to illustrate the RRT Integration test stage uses of six test agents instead of five.

2.2.3 Exact Current Flaky Test Prevention Methods

Exact employs several methods to mitigate flaky tests. One particularly interesting approach involves deploying multiple instances of templated data. Before describing this intricate method, several more straightforward techniques are explored.

Common methods employed by Exact to prevent flaky tests

Exact employs several methods to mitigate flakiness. These include common practices such as retrying failed tests, utilizing dedicated database instances for each test agent to minimize resource contention and data inconsistencies, and standardizing the test environment to reduce platform-specific issues. We discuss these in more detail in this Section.

Retrying Failed Tests: Failed tests are rerun on the same or a different device, with a maximum of 3 attempts per test. This approach can be effective in mitigating flakiness caused by concurrency, randomness, resource leaks, and test order dependency. However, excessive reliance on retries may decrease the urgency to address the underlying issues, potentially leading to an accumulation of flaky tests.

[3] https://github.com/microsoft/testfx

[4] https://specflow.org/

Dedicated Database Instances: Each test agent for API or integration tests has its own dedicated database instance. Tests on each test agent are executed sequentially, preventing test interference such as contention for database resources or inconsistent data due to read uncommitted isolation levels. This approach aims to mitigate the most common cause of flakiness: concurrency issues [23].

Standardized Test Environment: As described in Section 2.2.2, Exact standardizes its test environment by using a single type of test platform. This consistency in hardware and software configuration helps prevent flakiness caused by platform-specific issues [23, 94].

Temporary Test Deactivation: Unlike Google [70], Exact does not have a dedicated team solely focused on identifying and addressing flaky tests. Teams at Exact are responsible for investigating individual tests that cause RRT pipeline failures. To prevent missed releases due to flakiness and mitigate the potential for shared responsibility, which can exacerbate flakiness [68], such tests are prioritized for classification as either a system fault or a flaky test. This prioritization allows for a more timely initial assessment without immediately burdening the responsible developer with a required fix. If the test is classified as flaky, it is temporarily deactivated until it can be fixed.

Multiple instances of templated users provide natural database test-isolation

Another more unique and sophisticated strategy Exact employs to mitigate flakiness is to populate the database with multiple instances of user archetypes created from several templates, following the *Make Resource Unique* refactoring technique [21]. This configuration primarily aims to speed up and simply tests while also preventing flakiness.

Tests can retrieve and manipulate the data of these user archetypes, reducing both implementation complexity and execution time by preloading all data instead of generating it dynamically. The essential detail is that there are multiple instances of each template, such that tests can request a specific user archetype, and the framework guarantees the return of an unused instance. This eliminates the need for tests to perform explicit cleanup operations after manipulating user-level data, by providing natural database test isolation.

A background process handles repopulation to prevent underpopulation of mock instances. If the number of instances of an archetype drops below a certain threshold, a background service is initiated to create additional instances. This implementation detail has potential side effects, which will be discussed in Section 3.2.1.

This approach of *multiple instances of templated data* has limitations, however. It is not universally applicable to all database data. In certain scenarios, it can lead to excessive data growth. For instance, when applied to chained many-to-many relationships, an exponential number of instances per chain link is required, rendering it impractical to instantiate duplicates for all data in the database.

2.2.4 Motivation to Address Flaky Tests

Exact describes several key motivations for addressing test flakiness within their organization, which are the following:

Economic: Flaky tests have a significant economic impact. Each pipeline run incurs a substantial cost, with estimates suggesting that a single pipeline run can cost upwards of 10 euros in server costs. Initially nearly 3 out of 4 pipelines at Exact fail due to flakiness and need to be rerun, this would indicate that on average an upwards of 30 euros is spend on flaky tests every FRT.

Developer Sentiment: Flaky tests negatively impact developer sentiment and efficiency. Developers at Exact discredit non-flaky test failures, and experience frustration and wasted time when dealing with intermittent test failures, hindering productivity and morale.

Business: Exact experienced missed planned release deadlines potentially caused by flakiness-related pipeline failures. This impacted business operations, disrupted project timelines and potentially impacted customer expectations.

2.2.5 General Testing Issues

Exact faces several challenges that are not unique to Exact and affect test flakiness. We denote some of these fundamental issues for you to easier relate other projects to Exact.

Legacy code: Exact's long history, spanning over 40 years for the core product and 19 years for Exact Online, has resulted in a complex codebase with legacy design decisions that continue to influence today's development.

Many Contributors: With over 2,200 employees, including over 600 full-time equivalent (FTE) engineers – more than 250 of whom are dedicated to Exact Online – distributed across multiple continents, Exact Online experiences a high volume of code changes with diverse coding styles. This makes it challenging to maintain, communicate, and verify the intended uses for certain (testing) frameworks.

Various skill levels: Exact employs developers with a wide range of experience, from newcomers to industry veterans. Less experienced developers may lack a deep understanding of implementation details, specific standards or frameworks and may be more prone to making structural mistakes or writing smellier code.

Randomized test execution order: The randomized execution order of FRTs, as explained in Section 2.2.2, complicates the replication of specific test execution sequences and the identification of flakiness related to ordering issues.

Server dependencies: Exact Online is a .NET cloud application. The benefit of .NET cloud applications is that they only have to be developed for one type of deployment, preventing platform-dependent flakiness, and allow for quick shipment of updates. However, .NET cloud applications often rely on external services, such as databases, possibly resulting in concurrency or unavailability issues.

In the next chapter, we describe how some of these general testing issues manifested as flakiness at Exact. We then formulate three research questions, each accompanied by a corresponding hypothesis, based on the current situation at Exact and findings from related work described in this chapter.

Chapter 3

Approach

This chapter discusses the objective of *addressing flakiness in database-reliant industrial systems* within our case study [90] of flakiness at Exact. Our study centers around quantifying the extent of flakiness at Exact, which we determine through a flaky test rerun approach to provide us and the developers at Exact with quantitative results of the test flakiness. Together with the developers and based on our initial findings, we determined the root causes of flakiness, which we then used to formulate our three research questions (RQs). For each RQ, we investigate the impact of a different flakiness mitigation method applicable to a different type of flakiness.

Sections 3.1.2, 3.1.3, and 3.1.4 highlight the motivations, Exact-relevant technical details, and related academic work, behind each of the different automatic flakiness-mitigation strategies that we use to answer all three research questions. Section 3.2 explains the tools and mitigation strategies employed to address each question and the experimental setup used in this study.

3.1 Main Objective

This research aims to address the significant challenge of flaky tests in large-scale, database-reliant industrial software systems. While there has been growing interest in flaky test research, much of the focus has been on open-source projects, with less attention given to the unique challenges faced by typical industrial systems, as evident in the previous Section 2.1.

Not only do large and globally recognized companies such as Apple [53], Google [71, 72, 121], Microsoft [43, 55], Meta [7, 65] and Spotify [78], suffer from significant server costs and inefficiencies due to flaky tests. So do a broader range of companies, with over 97% of developers regularly observing flaky tests [79]. Certain industrial scenarios are under examined, leading to situations such as at Exact, where the engineers find that they lack readily applicable solutions to address flakiness.

This study addresses that research gap by specifically investigating and addressing the flaky test situation at Exact. The overarching goal is:

***Goal**: Addressing flakiness in database-reliant industrial systems.*

3.1.1 RQ 0 - Flakiness Discovery

To address flakiness at Exact, we first needed to identify flaky tests. We briefly explored multiple literature-inspired approaches to identify flakiness (outcomes reported below) and then ultimately decided on designing a rerun strategy to provide a more manageable, complete, dated, unbiased, and realistic set of flaky tests. While this technique has been employed in multiple related studies (as discussed in Section 2.1.4), we implemented the approach with several key design decisions and data-wrangling steps to enable the measurement and analysis of the practical exhibition of flaky tests with deeper insights.

The following sections explain the insights gained from the other three literature-inspired approaches: scouring commits [64, 118], leveraging developer notions [23, 81], and utilizing existing sets [92, 121]. They describe the quality attributes associated with our rerun approach and how these attributes influenced key design decisions and important data-wrangling steps. Lastly, these sections highlight the differences between our rerun approach and those employed in other academic work. These differences allowed for a more practical and comparative set of flaky tests, which, in turn, motivated our three research questions.

Alternative Solutions for Establishing the Flaky State at Exact

Scouring commits or threads for terms similar to 'flaky test' resulted in a limited set of flaky tests based on developer assumptions. The term 'flaky' appeared in communication threads roughly once a week; following these leads yielded a limited set of flaky tests. In most cases, the search hits for 'flaky' were related to time-sensitive failing RRTs, which often led to quick test disabling or retries. These decisions were typically based on the local build successfully executing the tests. Ideally, such situations should be followed by backlog items prompting proper investigation and root cause determination. However, based on the 600+ (2%) ignored tests, some dating back several years, this process is not always followed consistently. These ignored tests could potentially provide a valuable starting point for further research; however, most were outdated and would require significant effort to update to the current codebase, database schema, or other changes, with no guarantee they would still exhibit flakiness.

Developer notions regarding flaky tests provided us with various insights, some of which were partially correct. We informally questioned several developers with varying experience levels from different departments to gain insights into the current state of flakiness at Exact. Consistent with existing literature, most developers acknowledged flaky tests as a significant nuisance they encounter regularly. Opinions regarding rerun strategies were divided. A common belief was that flakiness within UI tests had the biggest impact on flaky pipeline failures. However, our rerun approach revealed this assumption to be incorrect. While there are generally more flaky UI tests compared to other test types, they exhibit flakiness less frequently. No UI tests had a pass rate lower than 0.9, making Exact's existing rerun strategy a very effective mitigation technique, keeping the Flakiness Pipeline Pass Percentage (FPPP) of the UI test stage above 90%. Other specific assumptions regarding flaky tests, such as flakiness due to improper resource disposal, turned out to be valid.

Exact's current set of known flaky tests was excessively large and showed that almost every test has exhibited flaky behavior once over the time of all its executions. Exact employed

the automatic flaky test detection tool from Azure DevOps[1]. This resulted in a vast majority of tests being flagged as flaky within the main branch. This large set of flaky tests, with an overabundance of sporadic flaky tests (i.e., tests with pass rates higher than 0.9), was challenging to use for determining the true causes of flakiness. Additionally, this approach was not suitable for accurately tracking changes in flakiness within the system, as it is difficult to control for the number of test reruns. This data set relies on CI pipeline runs; therefore, fixing the number of reruns and the initial set of flaky tests essentially replicates the rerun approach with less control.

Our hypothesis was, and later confirmed by our observations (Section5.2.4), that the overabundance of flagged flaky tests stem from environmental flakiness. In essence, rerunning a test often enough can eventually lead it to fail due to factors outside the scope of the test. This phenomenon causes every test to eventually appear flaky (Sec 5.2.4), an observation that coincides with Facebook and [65] and Gao et al. [30] its assumption that "*all test are flaky to some degree*", and somewhat with the findings of Alshammari et al. [4], who observed that 55% of their identified flaky tests exhibited flakiness only after the hundredth rerun.

Identifying Flakiness with a Same-Commit Rerun Approach

One study goal is to accurately and comprehensively model the realistic flaky state of the system in a repeatable manner with manageable costs. It needed to showcase the actual impact and either prove or disprove the preconceptions of the developers regarding flaky tests. To achieve this, we conducted **benchmark runs that repeatedly executed the entire CI pipeline on the same commit 11-36 times** (Section 3.2.4). This approach incorporates several key considerations:

- **Accuracy**: We opt for reruns instead of relying on tools finding factors related to flakiness [10, 37, 38] or existing sets of known flaky tests [92, 121] to minimize the number of tests that can only potentially behave flakily but do not do so in practice or do so only very rarely.
- **Completeness**. To ensure comprehensive coverage, we chose to run the entire active test suite multiple times. This approach helps prevent overlooked flaky tests that might be missed when only rerunning known flaky tests, a common limitation in other academic work [56, 58].
- **Realism**: We execute all tests through the normal CI pipeline stages on the same machines used in the production-development workflow. This approach simulates the actual flaky state of the system, considering factors such as machine-specific variations and environmental influences, thus negating the impact of localized faults as observed in other industry research where 86% of tests exhibited non-repeatable behavior when executed on different machines [55].

[1] `https://learn.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management`

- **Comparability**: To ensure comparability, we execute all runs within a benchmark on the same commit with the same configuration and run all benchmarks on the same day of the week, during periods of minimal resource congestion, and on similar hardware. This approach limits confounding factors thereby attempting to increase repeatability and comparability, a challenge common in other research efforts [27, 30, 55, 94].
- **Manageable Costs**: We execute the pipeline between 11 and 36 times for each flaky benchmark. We observed diminishing returns in terms of the number of newly identified flaky tests with each subsequent rerun. We determined that 26 runs were sufficient to achieve a stable measurement, with pass rates not shifting by more than 0.1 and only a consistent amount of new flaky tests being identified. While other approaches utilize 100 or more repeated test executions [4, 55, 58], we do not have the capacity to execute such a large number of runs for the entire pipeline multiple times for each of our three research questions, given that each research question requires at least two benchmarks, and conducting 100 runs per benchmark would be prohibitively expensive.

To showcase the actual impact and either prove or disprove the preconceptions of the developers regarding flaky tests, we analyzed the raw data obtained from the benchmark. Section 3.2.4 discuss how we aggregated this data and compared flaky-state measurements in more detail, along with all observed measured metrics. Here, it is important to understand that we categorize test failures based on their test-failure message. **This categorization of test failures based on their test-failure messages serves as the foundation of this research by facilitating data-grouping**, allowing us to identify patterns and trends in the exhibiting flakiness.

This initial classification was performed without relying on internal knowledge, using only the error messages from flaky tests identified through the rerun strategy. Later, we refined, merged, or deleted a small group of categories based on discussions with Exact developers. The involvement of Exact developers in the data-aggregation process and how their preconceptions, combined with the data, led to the formulation of our three research questions is discussed in the following sections.

Leveraging Exact's Knowledge in the Data Aggregation Method

Exact not only provided data for this research but also actively collaborated in the analytical process. This subsection describes how we engaged in a cyclical process, leveraging the developers' knowledge and experience of the codebase to further categorize and analyze flaky tests.

We initiated the process by informally questioning developers from various departments and seniority levels about their perceptions of the origins of flaky tests. Concurrently, we executed multiple pipeline runs, each linked to the same commit, which represented their most recent release at the time. This approach allowed us to analyze a commit that was considered fully functional and had already passed a pipeline run.

Subsequently, we examined all failed attempts and identified common categories based on common phrases within error messages. We grouped these failed test attempts into the aforementioned categories using pattern matching techniques.

The identified categories, together with other relevant metrics, were visualized and presented to Exact developers. Together with these developers, we more deeply analyzed the categories,

refining them and introducing new ones. The additional metrics included the tests with the highest number of failed attempts, execution times, test ownership, VM information, and more.

We iteratively repeated this data analysis multiple times. With each iteration, new and clearer patterns emerged, enabling us to formulate hypotheses about the root causes of flakiness and assist developers in fixing individual tests.

We Focus on API and Integration Tests Within Exact

Our research regarding flakiness at Exact focuses exclusively on API and Integration tests. These test categories are, based on our initial benchmarks, the most problematic and exhibit the least diffused flakiness. The initial preconception at Exact was that UI tests might be the most problematic due to their high frequency of failed attempts. However, further analysis revealed that they rarely resulted in actual failing tests (defined as tests with three consecutive failed attempts). Failing tests and failing pipelines are according to Exact considered the worst impact of flakiness. Consequently, with respective FPPPs of 60% and 40%, Exact considers the API and Integration test stages to be more problematic than the UI test stage, which exhibits an FPPP of 90%.

We excluded UI tests from our experiments due to the significant added complexity and limited anticipated benefits. Given the high FPPP of UI tests, incorporating them would provide limited insights compared to the potential gains from investigating API and Integration tests. Furthermore, extending our experiments to include UI tests proved challenging due to the distinct test setup employed for UI testing. UI tests utilize a different test framework, and the test stage employs 26 test runners that share the same database. These differences in framework and database utilization would have significantly increased the complexity of investigating the root causes of flakiness observed within API and Integration tests. Moreover, our initial investigation did not provide evidence to support the assumption that the root causes of flakiness within API and Integration tests would be applicable to UI tests.

Tests other than API, UI, and Integration tests do not exhibit significant signs of flakiness. Test stages such as Unit tests do not employ a test rerun strategy. While some flakiness might occasionally manifest within these other test stages, its impact is insignificant compared to the flakiness observed within the API, Integration, and UI test stages according to the engineers at Exact and our findings.

While all API and Integration tests were evaluated within our experiments, not all tests were adapted by the automated flakiness mitigation approaches mentioned in Section 3.2. We excluded tests generated using the SpecFlow framework due to their distinct framework and the associated challenges in integrating our tools. These SpecFlow-generated tests, as explained in Section 2.2.2, constitute 1-19% of the test suite. The SpecFlow framework presented challenges in inserting hooks into the code for our tools. We therefore decided to address this aspect in potential future work, after establishing the effectiveness of our methods in improving flakiness within the existing test suite.

We Developed Flakiness Mitigation Approaches for Root Causes of Flakiness at Exact

Based on our initial results, we identified that the majority of flakiness at Exact stemmed from incorrect test data or the unavailability/absence of test data. In collaboration with Exact developers, we reduced these issues to three root causes: redundant database (DB) background tasks, implicit disposal of test data, and dirty tests. We developed an automatic flakiness mitigation approach for each of these three root causes and formulated three research questions to investigate their impact on test flakiness at Exact. These are discussed in more detail in the subsequent three Sections 3.1.2, 3.1.3 and 3.1.4.

3.1.2 RQ 1 - Redundant Database Background Tasks

For our first approach to automatically reduce flakiness at Exact, we investigate the effect of minimizing redundant database background tasks. This Section shows why redundant background tasks are a root cause for flakiness at Exact, how this correlates to other research, and that we expect to solve 20-60% of flakiness by minimizing these background tasks.

Exact Utilizes Production Databases Filled with Mock Data For Regression Testing

Exact's current testing framework utilizes slightly more than 10 databases, each containing over 1,500 tables. These databases are replicas of the production database populated with mock data. This approach aims simulates the production environment, and therefore **includes unnecessary configuration options and background tasks that are irrelevant for the test environment**. For instance, cleanup processes or scheduled updates, although crucial for the production database, are redundant within the testing environment. At the start of each test, a fresh, clean version of the database is spun up and remains online for a maximum of 210 minutes, according to Exact's current configuration, before being discarded.

Exact Intertwined Performance and Regression Testing

This decision to create a complete copy of the production database stemmed from a design choice to precisely mimic the production environment. This approach intertwines the responsibilities of performance testing and regression testing; Performance tests are responsible for verifying reliability, while regression tests ensure system functionality [44]. This integration, while potentially effective in catching realistic load-induced bugs, also introduces background tasks into regression tests, causing load variations and data unavailability, which can contribute to flakiness [73, 94].

Initial Benchmarks Indicated Concurrency Issues at Exact

Our initial exploratory research suggested that most test attempt failures originated from database unavailability, potentially due to locked tables or missing data. While missing data could be a contributing factor, locked tables appear more likely, as tests often pass on subsequent reruns using the same or different databases. However, lock congestion issues can only occur due to

asynchronous processes, since within Exact's API or Integration test stage, each test runner operates with its own dedicated database and executes tests sequentially. These asynchronous processes can originate from within a test (related to the 'Fire and Forget' code smell [31]) or from scheduled database background tasks. Based on discussions with Exact developers, we believe that the 'Fire and Forget' code smell is less prevalent in their testing environment. Therefore, do we focuses on the latter, namely lock congestion caused by scheduled database background tasks. Which can be caused by Exact's design decision to replicate the production environment, including all its database maintenance tasks, within the testing environment.

We Investigate DB Task Minimization Based on Flaky Test Benchmarks at Exact

The Exact developers, based on our initial benchmark, suggest that flakiness is caused by redundant tasks. While other work have acknowledged concurrency as a significant contributor to flakiness [23, 55, 64], they do not explore solutions that re-evaluate the entire test environment configuration. Some research incorporates the test environment to identify flakiness through environmental perturbations [22, 73, 93, 100] or evaluates concurrency bugs in Android or GUI applications [2, 22, 61], but none assesses the impact of redundant concurrent processes. Therefore do we pose the following research question:

***RQ 1**: How does minimizing DB background tasks impact test flakiness at Exact?*

Hypothesis RQ 1: 20-60% Expected Flakiness Reduction through DB Background Task Minimization

By minimizing redundant background tasks, we expect a significant reduction in flaky tests by resolving most database-related data unavailability issues. While other studies suggest that concurrency is responsible for 20-26% of flaky tests [23, 64], our initial benchmark suggests it might be considerably higher.

Through discussions with Exact developers, we found that our error message categories – 'Access Denied', 'Lock Request', 'Login', 'Timeout', 'User Not Active', and 'Value Null' – might all stem from concurrency or data unavailability issues. These categories are quite prominent within our initial flaky test benchmark. We observed that:

- 76% and 47% of flaky API and Integration tests, respectively, failed at least once with error messages belonging to one of the aforementioned categories.
- For test-attempt failures, 72% and 55% of failed API and Integration test attempts, respectively, belonged to the aforementioned categories.
- Tests related to these categories were responsible for 62% of failed pipelines.

These numbers from the benchmark represent an upper bound for the potential number of flaky tests that might be resolved through DB background task minimization. The actual effects of DB background task minimization are likely to be less significant. We expect to see a reduction in flakiness within the system ranging from 20% to 60%, with similar improvements in the number of failing pipelines. This represents a wide range, but regardless, we anticipate a significant reduction in test flakiness.

3.1.3 RQ 2 - Implicit Disposing

For the second approach to automatically reduce flakiness at Exact, we investigate the effect of making implicit disposal of test data explicit. This Section shows why *Implicit Dispose* is a code smell and a root cause for flakiness at Exact, how this correlates to other research, and that by explicitly disposing of test data we expect to solve 13% of flakiness.

Exact Utilizes Templated Data Entries For Test Data Isolation

Exact employs multiple copies of templated archetypes in data entries to isolate data manipulation. These one-time-use entries allow testers to avoid manually setting up and cleaning required data. This approach adheres to the *Make Resource Unique* refactoring technique to prevent the *Test Run War* smell – which arises when a test allocates resources that might be used by other tests – by ensuring each test utilizes sperate data entries [21].

However, this method becomes impractical for data with required *one-to-many* or *many-to-many* relationships, as the number of data entries grows exponentially with the relationship depth. Consequently, Exact limits this method to one or two levels deep, relying on persistent data and manual cleanup by developers for deeper relationships.

Exact its Use of the .NET Dispose Pattern for Database Rollbacks

To automate cleanup of direct modifications, Exact utilizes read uncommitted[2] transactions and transaction rollbacks[3] within the `Dispose` method of the `IDisposable`-inheriting connection object. The `IDisposable` interface signifies the use of Microsoft's .NET Dispose pattern[4], designed for handling unmanaged resources like database connections [18].

This automatic cleanup partially adheres to the *Setup External Resource* refactoring technique applicable to the *Mystery Guest* and *Resource Optimism* smells which are present when a test access external resources or when it does so without checking their availability, respectively [21]. However, Exact's test framework only implements the cleanup aspect of the refactoring, not the explicit creation and allocation parts. Nevertheless, Exact's test framework mitigates three test smells known to result in flakiness [4].

Implicit Disposal of the Connection Object Can Result in Dirty Data

The automatic cleanup only occurs when developers strictly follow Exact's specific test framework. Implying that it only happens if: developers refrain from committing their data, use the predefined DB connection object, and explicitly dispose this connection object. In practice, this adherence is not always perfect, and we aim to investigate and quantify the impact of such de-

[2]Read uncommitted docs: `https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql`

[3]Transaction rollbacks docs: `https://learn.microsoft.com/en-us/sql/t-sql/language-elements/rollback-transaction-transact-sql`

[4].NET Dispose design pattern: `https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose`

viations by addressing the next two research questions. RQ 2 focuses on the last of the three requirements for automatic cleanups: explicit disposal.

Initial Benchmarks Suggest Over 25% of Remaining Flakiness at Exact Stems from Implicit Disposal

After DB-task minimization, we presented the new flaky-test findings to Exact developers. They examined the tests that consistently failed on the first attempt, which we refer to as 'always first-attempt-failing' tests. These tests were also the most frequently failing ones. Through manual examination of the 39 always first-attempt-failing tests, developers identified that 11 stemmed from 4 tests that neglected to explicitly dispose of an `IDisposable` object. These 4 tests leave the database in a dirty state. Some dirty data remained until the end of the test suite, while other DB data was cleaned at an unexpected time when the garbage collector cleaned the `IDisposable` connection object.

Implicit Dispose is a Code Smell Causing Nondeterministic Behavior

The Dispose pattern is not unique to Exact; it is a widely used .NET design pattern introduced in 2005 and discussed in many resources [6, 18, 42, 109, 115]. The `IDispose` tag has over 1,400 related StackOverflow questions, with the top question viewed nearly half a million times [77].

The Dispose pattern, when implicit disposal occurs, is inherently nondeterministic. Brian Pepin and Joe Duffy warn about the potential for nondeterministic bugs arising from implicit disposing in the book on the .NET Framework Design Guidelines [18]. For this reason, **we consider implicit dispose to be a code smell**.

The Implicit Dispose Smell Can Exist with All Persistent Data

The **Implicit Dispose** smell is not limited to test code or database-related data. For instance, a `FileStream` object also inherits from `IDisposable`, and when implicitly disposed, it can lock a file until a nondeterministic finalization event occurs [18]. This `FileStream` object interacts with the regular file system and is applicable in both production and test code. We observed non-db-related Implicit Disposal smells causing flakiness within Exact. Through one of our initial methods of scouring issue-threads within Exact with regard to flaky tests, we found two instances of a test behaving erratically due to an undisposed object persisting within a singleton[5]. This demonstrates the presence of the Implicit Dispose smell within (essentially) static variables at Exact. This is significant considering that Zhang et al. [119] found that most test-data dependencies stemmed from shared static variables.

Current Dirty Test Solutions Are Not Applicable for Implicit-Disposal Bugs

A dirty test itself is not inherently flaky. Only when a subsequent test[6] executes with 'dirt' persisting from a prior test can flakiness arise. For this reason, dirty tests are frequently associated with order-dependent flakiness [29, 37, 46, 56, 110].

[5] For the unbeknownst reader of singletons consider them to be static global objects.

[6] A second test can also be the same test executed a second time [113].

Academic work builds on this principle of order-dependent flakiness by leveraging the test order to identify flakiness [56, 119]. However, these methods are ineffective in capturing all flakiness caused by the implicit-disposal smell. The implicit-disposal bug entails keeping data dirty until the garbage collector non-deterministically cleans it up in a concurrent process. Due to this non-determinism and concurrency, implicit-disposal dirty tests can cause flakiness with a fixed test order while also being order-dependent.

Other approaches tackle dirty tests by identifying dependencies between tests [46, 56, 80, 119] or in-memory pollution [37], with the latter approach not detecting dirty external memory, such as database data. To illustrate the issue with the former, consider the scenario where a nondeterministic dispose triggers a database rollback, locking tables while another test is executing. Thereby indicating that any two tests that share the same table, share dependencies. Consequently, depending on whether dependency-detection-methods consider tests that share a database-table as dependent, will produce either an excessively broad set of potential flaky tests or miss potential connections causing flakiness.

Current Automatic Implicit-Disposal Prevention Methods Are Incomplete

The implicit disposing code smell lacks a fully functional solution. Some solutions exist within grey literature, describing refactoring techniques [45] or mentioning Microsoft's code analysis rule CA2000 to detect the smell [112]. The former approach would significantly complicate Exact's tests and necessitate a substantial refactoring effort. While Microsoft's code analysis rule is a potential solution, as discussed in Section 3.2.2, the rule is outdated, neglected, computationally expensive, and inapplicable to the specific smell instances most prevalent within Exact [40, 52].

We Investigate the Impact of the Implicit Dispose Smell

As stated, implicit disposal can lead to persistent data remaining dirty until it is concurrently cleaned up by the garbage collector at an undefined time, potentially causing unexpected bugs in both production and test code. Although implicit-disposal bugs are a recognized problem, prevention methods are outdated and incomplete due to their computational complexity. Based on our initial benchmarks Exact developers identified that 11 out of the 39 most frequently failing flaky tests resulted from dirty tests caused by implicit data disposal.

While the academic literature describes several prevention and identification methods for most dirty tests, implicit database disposal eludes these methods due to its inherent non-deterministic and concurrent nature. With the implicit-data-disposal smell unexplored in literature and our initial examination suggesting it as a significant contributor to flakiness, we pose the following research question:

***RQ 2**: How does explicitly disposing test-data impact test flakiness at Exact?*

Hypothesis 2: 13% Expected Reduction of Flakiness through Explicit Test-Data Disposal

We initially expected implicit data disposal to be infrequent within tests due to the clear ownership of disposable objects within tests. This is in according with Duffy stating that explicit

disposal becomes challenging with ambiguous ownership [18], and given that ownership of an object within tests should be unambiguous.

We anticipate that most implicit data-disposing instances will contribute significantly to flakiness. This is supported by our initial findings where only 4 implicit data-disposing instances were responsible for 11 flaky tests.

Other studies have shown that 'Test Order Dependency' and 'Resource Leak' can be responsible for 8-11% and 5-7% of flakiness, respectively [64, 23]. Both these categories of flakiness can stem from implicit data disposing. According to the literature, it can therefore be responsible for 13-18% of flaky tests.

Based on our initial results, we posit that implicit disposing is responsible for more than 13% of flaky tests. Our post-'Minimized DB background tasks' benchmark indicated that implicit data-disposing might be responsible for over 25% of the remaining flaky tests. Given that DB background task minimization is expected to resolve 20-60% of the flakiness, we would assume that implicit disposing is responsible for 10-20% of the flakiness. However, due to a bias in the observed set of tests, we expect the actual number to be much higher. We only examined always first-attempt-failing tests. Meanwhile, we assume that flakiness due to implicit data-disposing would mainly exhibit within varying tests because of implicit disposing's nondeterministic nature combined with the fact that Exact employs varying test orders. Therefore, we expect to see more flakiness in the entire set relative to the examined subset.

3.1.4 RQ 3 - Dirty Databases

For the third approach to automatically reduce flakiness at Exact, we investigate the effect of DB-dirty tests by instrumenting tests with a DB sanity check and subsequently disabling dirty tests. This Section shows why tests can become dirty within Exact, how prevalent this is as a root cause for test flakiness at Exact, and that we expect to not only solve flaky tests but also break brittle tests by disabling dirty tests.

Exact's test framework relies on database transaction rollbacks to reset persistent data between tests, as discussed in Section 3.1.3. Similar to findings from industry practice [55], these test-cleaning frameworks are not infallible and are susceptible to bugs or misuse. RQ 3 aims to encompass all scenarios where bugs or human error lead to dirty database data. This section illustrate these issues and how other researchers have addressed them.

Malpractice by Exact Developers Can Cause Dirty Database Test Data

Transaction rollbacks within Exact's tests occur only if developers adhere to three steps: not committing data, using the predefined DB connection object, and explicitly disposing of this connection. RQ 2 investigates the last step in detail. Here, we take a broader approach to capture any violations of these three steps that might lead to dirty databases. For instance, if a test directly connects to the database for data manipulation instead of using the predefined framework object, no rollback occurs, thus potentially leaving the database dirty.

Bugs can also cause dirty data to persist within the database. An example involves a function initiating an inner transaction. Because Exact's test framework also starts a transaction, nested transactions arise. Unaware developers might encounter unexpected behavior when triggering rollbacks [5, 88]. When using nested transactions and a rollback is triggered, it rolls back to the top transaction instead of the inner one [14]. This causes any operations following the rollback to occur outside a transaction and persist in the database. See Code Listing 3.1 for an example.

```
1  create table #t (col1 int)
2  insert #t (col1) values (1)
3  BEGIN TRANSACTION
4  update #t set col1 = 2 -- This gets rolled back
5  BEGIN TRANSACTION
6  update #t set col1 = 3 -- This gets rolled back too
7  ROLLBACK TRANSACTION
8  update #t set col1 = 4 -- This is run OUTSIDE a transaction!
9  COMMIT TRANSACTION -- Throws error
10 select col1 from #t -- Returns 4
```

Figure 3.1: Example of a Rollback Within a Nested SQL Server Transaction (adapted from Stack Overflow [5])

Parallel processes combined with read-uncommitted databases are notorious for potential dirty reads. Exact employs a method of using one database per sequential test execution to mitigate this issue. However, Exact's tests can sometimes involve asynchronous processes. These processes can leave the database in a dirty state while the main thread's transaction is neatly cleaned.

Initial Benchmarks Suggest Dirty Databases

Our initial benchmarks suggest the presence of dirty database data. Through error message categorization, we identified tests failing due to 'Setting Configuration' or 'GL Accounts' errors. The 'Setting Configuration' error is particularly interesting, as some cases indicate that global settings differ from expected values. Global settings can affect nearly every test. If a test modifies these settings without restoring them, it can cause unexpected behavior for almost every subsequent test.

We Investigate the Effect of Database Dirtiness on Test Flakiness

Zhang et al. found that most test dependencies causing flakiness stemmed from dirty heap data [119], leading to the development of heap pollution detection methods [37]. However, due

to Exact's heavy reliance on databases, we assume that most dependencies reside within the databases. Therefore, we pose the following research question:

***RQ 3**: How does filtering database dirty tests impact test flakiness at Exact?*

Hypothesis 3: Expected To Break Brittle and Fix Flaky Tests By Disabling Database-Dirty Tests

Literature suggests that 8-11% of flakiness is related to 'Test Order Dependent' tests [64, 23], with more recent work finding this issue to be dominant in Python, responsible for 59% of flakiness [36]. Luo et al. also found that 47% of flaky test-order-dependent tests failed due to external resources. We believe that this percentage of flakiness due to external resources is likely much higher within Exact due to its heavy reliance on databases.

We expect that some tests will also pass due to other tests manipulating the database. These are order-dependent tests known as *brittle* tests. ***Brittle tests*** **rely on other tests to set a specific state and fail in isolation.** Brittle tests might not currently exhibit flakiness, but they likely will in the future or when the test order changes. Some projects contain more brittle tests than tests that fail because a test has polluted the database [92].

3.2 Developed Methods and Tools

This section discusses the tools and methods used and developed in this thesis. This section provides the foundation for the methods and tools employed in the experiment and describes their potential pitfalls and coherent mitigation strategies. It also explain our own 'WeDispose' code analyzer rule for finding and patching implicit disposing code and discusses its necessity compared to existing alternatives. Further, it presents code listings that illustrate concrete examples related to the primary causes of flakiness within Exact. Finally, this section aims to enhance the applicability of this research by providing a foundation to better understand the methods used and, consequently, the results discussed in Chapter 4.

3.2.1 RQ 1 - Minimizing Background Tasks

Minimizing DB background tasks involves identifying and stopping redundant tasks and reconfiguring the startup configuration to minimize the need for certain tasks. This process entails only configuration changes and does not require any modifications to production or test code. The primary changes include the removal of all data-cleaning processes and the reevaluation of the initial database population.

As mentioned in Section 3.1.3, Exact maintains multiple copies of templated data entries to allow tests to utilize unique copies. An automatic repopulation background process exists to ensure that the initial population contains a sufficient number of data entries. However, maintaining a buffer size to account for all potential worst-case scenarios would lead to significantly bloated databases. For example, Exact could require three data entries per test per template type, assuming that each test could theoretically use the same template and be rerun three times. This would result in over 54,000 database entries per template type within Integration tests alone.

Another factor to consider is a margin for the continuous addition of new tests. Exact reexamined the initial database population and lowered the thresholds for triggering the repopulation process to zero.

3.2.2 RQ 2 - WeDipose: Explicitly Disposing with a Roslyn Static Analyzer

We develop ***WeDipose***, a code analysis rule to find and rewrite all instances of implicit disposing. Finding objects that are both disposable and not explicitly disposed requires contextual information. To find all instances of implicit disposing, it is essential to know if a test class member variable is of type `IDisposable` and if no dispose call is made before that instance and any of its member variables are no longer necessary. We leveraged Roslyn for this, a compiler platform that provides a set of APIs and tools for code information within .NET applications written in Visual Basic as well as C#. Roslyn's extensibility allows developers to append their own rules to detect and refactor specific scenarios.

CA2000: An Ineffective Alternative for WeDispose

An existing Roslyn rule can find and explicitly dispose of implicitly disposed objects. However, this rule is not applicable for most cases within Exact. The .NET code analysis rule CA2000 is designed to find and fix the implicit dispose smell. However, for the following two reasons, this rule is not suitable for Exact:

Performance Issues: This rule has been disabled for over three years due to performance problems [40]. According to official GitHub issue-threads, "CA2000 must be disabled in any scenario" [52]. This rule is unusable, especially for large projects like Exact.

Incomplete Scenario Support: The rule is incomplete and does not support cases where the `IDisposable` instance is created through factory methods. Code Listing 3.2 shows the general setup of an Integration test that instantiates a disposable object through a factory method. As explained in Section 2.2.3, this is the most common way template users, used in almost all Integration tests, are instantiated within Exact's test framework.

Due to performance limitations and incomplete scenario support, we were unable to leverage existing detection methods. Therefore, we designed our own simplified version of the rule, called WeDispose, to handle most disposal cases within a testing context.

WeDispose Relies on Branchless Methods

WeDispose works by checking the type associated with any assigned expression or declaration statement. If the type inherits from `IDisposable`, the execution scope is followed. If the instance is not explicitly disposed, the declaration statement is flagged. Every flagged implicit-dispose instance is automatically patched by WeDispose. This patch involves wrapping the `IDisposable` instance in a `using` statement or calling `Dispose` at the end of the scope or in the linked cleanup method.

WeDispose is a rule specifically designed for testing. It assumes branchless scopes and links test initialization and cleanup attributes. WeDispose is built on the simplified assumption that a test or a test setup is branchless, allowing for a computationally feasible rule compared to

```
1  [TestInitialize]
2  public override void TestInitialize()
3  {
4      base.TestInitialize();
5      _env = EnvFactory.GetProcessEnvironment(_testDivision, TemplateUser.CustomerA);
6     ...
7  }
8
9  [TestCleanup]
10 public override void TestCleanup()
11 {
12     ...
13     _env.Dispose() // This is an explicit dispose that is often forgotten.
14     base.TestCleanup();
15 }
```

Figure 3.2: General setup of an integration test at Exact.

CA2000. However, this comes with the downside of potential false positives when a branch occurs and a disposable is explicitly disposed of in all inner scopes of branches but not in the outer scope. There are other edge cases where WeDispose lacks functionality, but it is designed to prioritize no false negatives over false positives.

Manual Verification of WeDispose Test Refactorings

We manually review and fixed all incorrect automatic refactorings to compensate for missed edge cases. This manual review allows us to confidently claim that all but one implicit-to-explicit dispose refactors behave as intended.

To maintain the feasibility of the manual review, we limited our refactoring to consider only test code and did not address the cause of test flakiness due to implicit disposal of production code. According to Eck et al., this might mean that we miss 15% of flakiness due to 'Resource Leak' [23]. However, we leave the evaluation of the actual impact on test flakiness of the Implicit Dispose smell in production-code open for future work.

3.2.3 RQ 3 - Instrumenting DB Sanity Checks Before and After Every Test

We add database sanity checks before and after every test to compare the database states. We then flag and disable all tests that modified the database. All sanity checks are performed on the most frequently used *test-user* database. We implement two types of sanity checks: a generalized check and an Exact-specific check:

1. **Row counts**: The *dirty-row-counts* check is a general-purpose sanity check that compares the total number of rows before and after the test execution.

2. **Settings**. The *dirty-settings* check is an Exact-specific sanity check that hashes all 'global settings' to determine if they have been modified. Exact isolates data within its databases based on users and enterprises, but some settings are 'global settings'. These global settings are database values that might provide default values or facilitate partial migrations to new features. According to Exact engineers, tests are absolutely not allowed to leave these global settings dirty, as such modifications can influence the behavior of all subsequent tests.

The benefit of this instrumented DB sanity check lies in its adaptability. Exact allows for some database values to be modified, such as 'last updated timestamps' or log table entries. We identified these exceptions by initially flagging all dirty data and then discussing the overview with Exact developers. This collaborative process resulted in the exclusion of 12 tables from the dirty-row-count sanity check.

To instrument the tests, we insert the sanity checks before and after each test, by utilizing the test initialization and test cleanup methods within each test class (see Code Listing 3.3 for an example). We perform these tests to identify cleanups at the test level to easily pinpoint the violating test. However, since class initialization and assembly initialization methods are called before any test initialization, tests classes or assemblies that leave the database dirty in these stages might go unnoticed. Additionally, tests that asynchronously clean up their dirtiness might also go unnoticed or be flagged non-deterministically based on concurrency timings. We do not account for these edge cases in this research and leave them for future work.

```
1  [TestInitialize]
2  public override void TestInitialize()
3  {
4  +   SanityCheck.save(); // Saves the current DB row counts and hashed settings
5      base.TestInitialize();
6     ...
7  }
8
9  [TestCleanup]
10 public override void TestCleanup()
11 {
12     ...
13     base.TestCleanup();
14 +   SanityCheck.compare(); // Compares the saved values to the current ones
15 }
```

Figure 3.3: How we inserted DB sanity checks.

For the experiment, described in more detail in Section 3.2.4, we conducted four benchmark runs: the baseline, one with all dirty-setting tests disabled, one with all dirty-row-count tests disabled, and one with both dirty-setting tests and dirty-row-count tests disabled. We used only one pipeline run to determine all dirty tests. Due to the fact that sanity checks occur within the test cleanup, we could only evaluate dirtiness in tests where at least one attempt terminated normally.

3.2.4 Same-Commit Rerun Benchmark

We gather all our data by repeatedly running the pipeline multiple times on a single commit. We refer to this collection of pipeline runs as a '**benchmark run**'. For each research question, we conduct two or more benchmark runs: one to establish a baseline and others to assess the impact of the applied change. This approach mitigates the effects of unexamined confounding factors, such as changes due to the system being a live environment, which we discuss in more detail in Section 5.3.4. The only difference between a baseline benchmark run and a post-change benchmark run is the specific change implemented.

We conduct all these benchmark runs using the same process, configuration, and hardware employed within Exact's CI pipeline to accurately reflect the actual flaky test situation at Exact. However, to avoid interfering with Exact's workflow and minimize variability in flakiness caused by the effects of hardware interference and Resource-Affected Flaky Tests [94], we execute all tests on Sundays when machine usage is typically minimal. Nevertheless, we simulate a high load within each benchmark run by executing multiple pipeline runs concurrently within virtual machines on the same physical device, possibly resulting in resource interference.

In our experiments, we encountered tests that consistently failed due to our applied changes. We do not consider these '*always-failing*' tests as *flaky* within the benchmark and therefore filter them from the categorization graphs and the FPPP. However, these *always-failing* tests are still visible within the Sankey graphs that illustrate changes in pass rates. We present these graphs and benchmark results in the following chapter.

Chapter 4

Results

This chapter presents the results obtained from the flakiness benchmark runs at Exact. It first discusses general observations, how external factors impact test flakiness and cause *sporadic flakiness* at Exact, and how to interpret result graphs accordingly (Section 4.1).

Sections 4.2, 4.3, and 4.4 comprehensively address the research questions, examining both test suite-wide observations and individual test behavior. Each section analyzes the overall impact, categorizes errors, identifies outliers, and describes the behavior of affected flakiness. Each section concludes with an answer to its respective research question. These section state that Exact exhibited various root causes of flakiness, with Database (DB) background tasks emerging as the most significant contributor, responsible for 40% of flaky tests. Implicit disposal and dirty tests, while contributing to a smaller proportion of flaky tests, were found to impact brittle tests that relied on their unintended behavior.

Finally, this chapter discusses how Exact has gradually mitigated the impact of flakiness within its system since the start of this thesis. We demonstrate that Exact's Flakiness Pipeline Pass Percentage (FPPP), which indicates the proportion of pipelines passing without any changes, increased from 27% to 96%. This improvement resulted in a 1.5x increase in the monthly release rate, which has remained at a record high for three consecutive months.

4.1 General observations

This section presents general observations applicable to all results obtained from flaky test benchmarks, providing essential context for their interpretation. To begin, we observed that a significant portion of flaky tests exhibit 'sporadic' flakiness, characterized by infrequent attempt failures that appear in less than 10% of their attempts. Conversely, a smaller subset of non-sporadic flaky tests contributes disproportionately to overall test failures. The characteristics of sporadic flaky tests are further explored in Section 4.1.1.

Moreover, the real-world experimental environment introduced inherent variability between benchmark runs, exceeding the variations attributable to the changes under investigation. This variability stems from external factors influencing the benchmark runs. The impact of these external factors is discussed in Section 4.1.2, with the specific effects of platform changes detailed in Section 4.1.3. The final two sections, Sections 4.1.4 and 4.1.5, establish a foundation for the remaining results by clarifying data-filtering procedures applied to specific metrics and graphs, and providing guidance on interpreting Sankey graphs to mitigate the influence of noise.

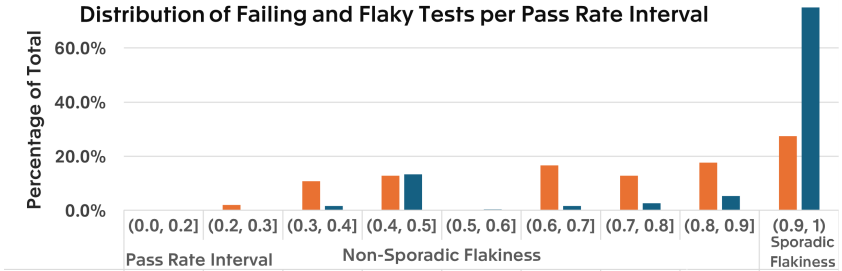
4.1.1 Sporadic Flakiness

Our analysis revealed that at least one-third of flaky tests within each benchmark exhibit *pass rates* between 0.9 and 1.0. We designate this group as *sporadic flaky tests*. The sporadic nature of these tests and the limited number of reruns in our experiments hinder the reliable assessment of behavioral changes. The distinction between *sporadic* and *non-sporadic flaky tests* enables more confident statements regarding changes in test behavior. We have employed this simplified differentiation instead of incorporating statistical confidence intervals because of the complexity of accurately establishing the latter for individual tests due to hidden dependencies between tests and other unknown confounding factors.

Existing literature has highlighted the multifaceted nature of flakiness, attributing it to various factors and acknowledging the presence of confounding factors that introduce hidden dependencies between tests [64, 23, 119, 9]. These hidden dependencies and confounding factors impede accurate statistical analysis of individual test flakiness. To illustrate the scope of this issue: Alshammari et al. observed that only 45% of flaky tests were identified within the first 100 runs [4]. Their work has likely overestimated the true percentage of flaky tests found within the first 100 runs, as other research referencing the same set of tests, has identified additional flaky tests unaccounted for by Alshammari et al. [56, 11].

The experiments in this study revealed that non-sporadic flaky tests have the most significant negative impact. Specifically, non-sporadic flaky tests are more likely to cause pipeline failures within the Exact Integration and API tests. Figure 4.1 demonstrates that 25% of all flaky tests across the benchmarks were classified as non-sporadic. Notably, this 25% was responsible for 72% of all test failures and 69% of all pipeline failures. In the default benchmark, consisting of 26 pipeline runs with a maximum of 3 test attempts, a test must exhibit at least 8 failures to be classified as non-sporadic.

It is important to note that the number of flaky tests, test failures, and their respective pass rates are calculated per benchmark and then aggregated. This approach accurately reflects the pass rates of a single benchmark release but introduces a bias toward the impact of frequently failing tests that have not been resolved when considering releases distributed over time. We consider this bias acceptable as it supports our primary objective to illustrate the relationship between pass rate intervals and the impact of flakiness.



	(0.0, 0.2]	(0.2, 0.3]	(0.3, 0.4]	(0.4, 0.5]	(0.5, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1) Sporadic Flakiness
Failed Tests	0.0%	2.0%	10.8%	12.7%	0.0%	16.7%	12.7%	17.6%	27.5%
Flaky Tests	0.0%	0.0%	1.7%	13.3%	0.4%	1.6%	2.6%	5.3%	74.9%

Figure 4.1: Distribution of failing and flaky tests per pass rate interval for all gathered benchmarks, representing how a small set of test is responsible for most pipeline failures

4.1.2 External Factors Impacting Test Flakiness

The benchmark runs occasionally experienced failures attributable to external factors generally unrelated to the tools or methods investigated in this research. External factors encompass any element outside the control of the test suite, such as offline services, resource contention between virtual machines (VMs), or bugs originating from external systems rather than from self-managed test or production code. While other works have investigated the impact and relationship of these factors [30, 94], this section focuses on the proportion and impact of such flaky tests within a real-world industrial context. Section 4.1.2 discusses the effects of unknown external factors, while Section 4.1.3 addresses the effects related to the known external factor, the test platform.

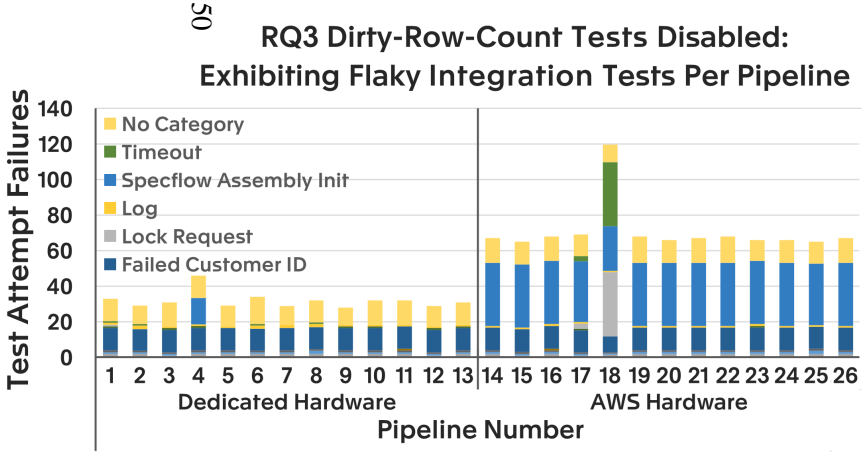
Test Runner Crashes

We encountered and filtered out pipeline runs where one of the test environments crashed due to bugs within the test framework. One such bug occurred when Exact newly integrated AWS to execute it pipelines. An uneven scheduling bug resulted in tests being assigned to already occupied test runners, leading to the failure of all tests assigned to those runners. Another bug was related to the cleanup processes on Exact's dedicated server. A test runner was observed to stop unexpectedly, halting all tests due to insufficient disk space on the physical device. These observations highlight how environmental issues can result in test failures. We filtered these pipeline runs from the results due to the clear and straightforward nature of the failures.

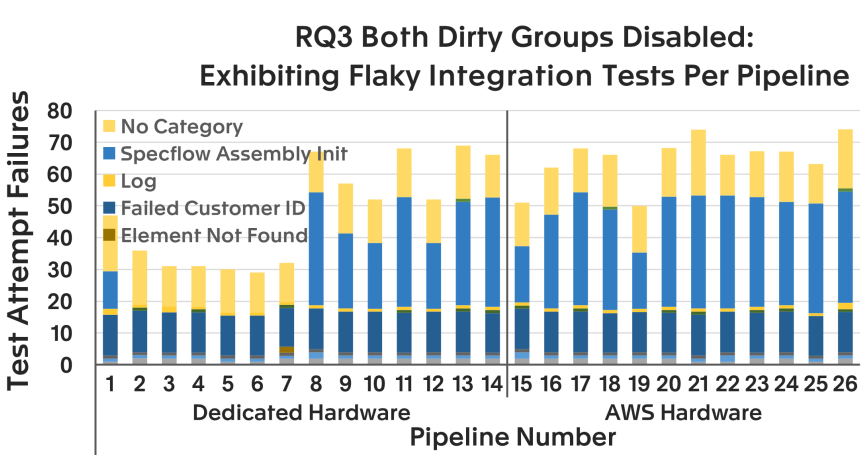
External Factors Causing Sporadic Flakiness

External factors induced sporadic flakiness within the test suite, manifesting as both clustered test failures and subtle individual test failures. However, pinpointing the root cause of individual test failures attributed to external factors and distinguishing them from other sources of flakiness, such as order-dependency, proved difficult. Consequently, a precise estimate of their proportion remains elusive. However, our manual inspection of several test failures revealed that some flakiness stemmed from unavailable external resources, such as network connectivity issues or unavailable remote services, and resource contention, such as slow database query responses resulting in test timeouts due to resource contention with other processes. The rerun strategy at Exact mitigated the impact of these external factors, as most tests passed on subsequent attempts.

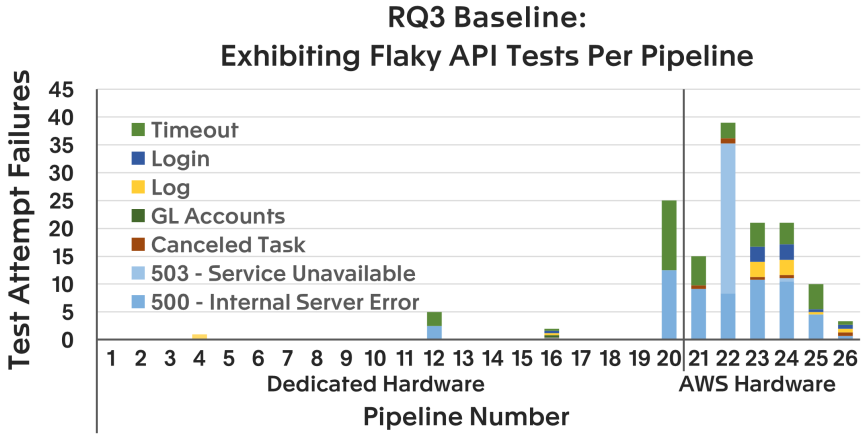
We observed that external factors can lead to clustered sporadic flakiness in approximately 1 out of 13 cases. This flakiness is similar to 'Infrastructure' flakiness observed by Gruber et al [36]. These clusters of flaky tests can be substantial in size and contribute significantly to upper outliers when comparing the number of flaky tests between repeated runs in one benchmark (see Section 4.5). This manifestation of clustered flakiness caused by external factors is visualized in Figure 4.2. Figure 4.2d reveals a disproportionately high number of flaky tests in pipeline runs 10 and 13 compared to other runs. All test attempt failures within these runs belong to the same category (i.e. 'Canceled Task'). This category indicates that the test received an abort command from the test suite. While the specific cause of this abort command remains undetermined, it is evident that it originates from an external source beyond the control of the test suite.



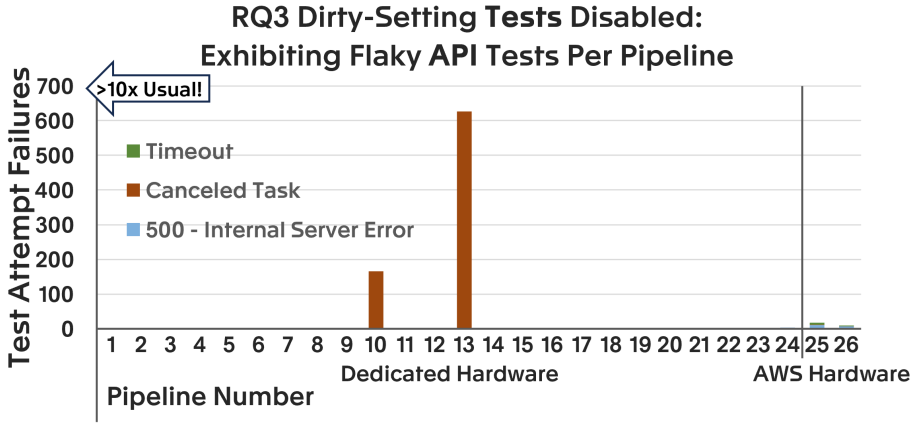
(a) More flaky test on AWS Hardware when running Integration Test with all dirty-row-count tests disabled.



(b) Overall more flaky test on AWS Hardware when running Integration Test with both dirty-row-count and dirty-setting tests disabled.



(c) More flaky API test on AWS hardware within our baseline benchmark.



(d) Two runs with significantly higher numbers of flaky tests exhibiting the same error within our API benchmark with all dirty-setting tests disabled.

Figure 4.2: Distribution of flaky tests exhibiting various categories and their associated categories per pipeline within the RQ3 benchmarks. Note: Since a single failure may belong to multiple categories, the category color distribution is proportional to the total number of categories and tests without assigned categories in that pipeline run.

Nonetheless, not all instances of clustered flakiness have a clear and equally sizable effect attributable to 'external factors', given that we observed various clusters of flakiness *potentially* caused by different external factors. While a comprehensive analysis of each instance is beyond the scope of this work, one additional cluster linked to platform changes is discussed in the next section. Furthermore, Section 4.2.2 explains four additional groups of flaky tests, potentially influenced by external or other contributing factors.

4.1.3 The Test Platform Impact Flakiness

Changes in the test platform increased the likelihood of certain types of test failures. Although this study aimed to minimize external factors, the partial migration of Exact's test platform to AWS necessitated the execution of RQ3 experiments on AWS servers. The only difference between the pipeline execution on Exact's dedicated servers and AWS servers is the underlying hardware and potentially the OS version, with the VM configuration remaining identical. Nevertheless, we observed a higher average number of flaky tests when running on AWS servers, manifesting as sporadic flaky tests for API tests and non-sporadic flaky tests for Integration tests.

Platform Changes Impact the Flakiness of a Specific Group of Tests

The impact of platform changes on the non-sporadic flakiness of specific tests is best illustrated by the 'SpecFlow Assembly Init' Integration test failures observed after disabling at least the row-count-dirty tests. Figures 4.2a and 4.2b visualize the Integration-stage benchmark runs after disabling the row-count-dirty and both-dirty groups of tests, respectively. These figures demonstrate a higher average number of flaky tests per pipeline when executed on AWS hardware compared to dedicated hardware. This difference is primarily attributable to the 'SpecFlow Assembly Init' tests, represented by the yellow portion of the bars.

Further analysis of the 'SpecFlow Assembly Init' tests within the 'Both Dirty Tests Disabled' benchmark revealed that a specific group of 36 tests across 5 assemblies consistently failed with the same error. These attempt failures cluster within an assembly, meaning that if one test attempt within an assembly fails, they all fail. This phenomenon occurred 12 to 19 times within each 'Both Dirty Tests Disabled' benchmark run. The root cause of this clustered behavior lies in an assembly initialization error, which stems from a check that verifies the online status of the locally hosted system intended for evaluation by the tests. Although this check should be independent of the database state, we observed that these AWS-influenced flaky 'SpecFlow Assembly Init' tests exhibit more consistent failures in benchmarks with at least the dirty-row-count test disabled compared to the baseline and dirty settings test disabled benchmarks.

Do not misinterpret this increased flakiness on AWS as AWS incompatibility. Many tests ultimately pass on AWS hardware, suggesting that the platform itself is not the primary cause. Furthermore, these failures are not consistent, as evidenced by the fact that only one of the five assemblies in the 'Both Dirty Groups Disabled' example failed on every first attempt. Most notably, this behavior is not exclusive to AWS. The same flakiness occurs on dedicated hardware, albeit less frequently. The small yellow spike at pipeline 3 in Figure 4.2b indicates the occurrence of 'SpecFlow Assembly Init' failures on dedicated hardware. The tests responsible for this spike on dedicated hardware are identical to those observed on the AWS platform.

Platform Changes Result in Increased Sporadic Flakiness

On average, and excluding occasional outliers, Exact's API pipelines executed on the AWS platform exhibited a higher number of flaky tests compared to pipelines executed on Exact's dedicated platform. This trend persisted across all RQ3 benchmarks with varying degrees of severity. The observed flakiness manifested within a constantly changing, and therefore sporadic, group of flaky tests. Most instances of flakiness were attributed to 'Timeout', '500 - Internal Server Error', or '503 - Service Unavailable' errors, as illustrated by the corresponding colors within spikes following test execution on AWS hardware in Figure 4.2c.

Most of the sporadic platform-dependent flakiness was attributed to timeouts during API calls. Pinpointing the true cause of API test failures can be challenging due to the limited contextual information provided by API calls in the event of an error. API calls often return minimal information regarding back-end issues, a design decision by Exact aimed at abstracting back-end details for customers. Nevertheless, manual inspection revealed that most '503 - Service Unavailable' errors were essentially 'Timeout' issues, indicating that the service simply did not respond within the allotted time instead of being genuinely unavailable.

Timeouts are unexpected given the sequential execution of tests, the use of a dedicated database, and the significantly allowed time compared to typical execution times. Potential explanations for timeout issues within Exact's API tests include resource contention from background services or intermittent connection issues to local services. Both scenarios could exacerbate flakiness when transitioning between test environments, contributing to the observed increase in sporadic flaky test failures.

4.1.4 Filtered Results

All graphs, with the exception of the pass rate graph, were filtered to include only flaky tests. The analysis of the changes introduced for RQ2 and RQ3 revealed brittle tests that consistently fail. These newly introduced, always-failing tests are not considered flaky and are therefore excluded from all graphs and the FPPP calculation. Note that these always-failing tests do not occur in the baseline benchmarks, as these represent Exact releases, which by definition cannot contain always-failing tests. A commit must pass all tests within a Release Regression Test (RRT) before it can be released (Section 2.2.1).

The Sankey graphs only depict tests that exhibit flaky behavior in at least one of the benchmarks being compared. Therefore, the set of tests within the pass rate interval [1, 1] (always passing) represents only a subset of all always-passing tests. As explained in Section 2.2.2, Exact encompasses approximately 25,000 Integration and API tests. Our results demonstrate that, within 26 runs, we typically observe flakiness in fewer than 250 of these tests.

4.1.5 How to Interpret the Sankey Graphs

Our Sankey graphs are sensitive to noise and observational randomness. Therefore, when interpreting the effect of the change between two benchmarks, one should largely disregard all flows into neighboring pass rate intervals. The Sankey graphs visualize the behavior of individual non-sporadic flaky tests. The Sankey graphs do not accurately depict whether pass rate changes of

sporadic flaky tests are induced by the actual difference between the two compared benchmarks because the data is polluted with sporadic flakiness arising from various other factors, such as order-dependent tests or external factors.

The sensitivity to sporadic flaky tests is best illustrated by the large shifts between the (0.9, 1) interval and the [1, 1] interval in Figure 4.12a. These large shifts within the intervals are attributed to the two large outliers visualized in Figure 4.2d.

Many non-sporadic flaky tests also exhibit jumps between neighboring interval bins. This behavior is likely due to the statistical nature of the results and the binning process. Values can be assigned to a bin even if they lie on its boundary. Preventing random shifts between bins would necessitate an infeasible confidence interval of 100%. While additional pipeline runs within the benchmark could mitigate these effects, it could also exacerbate the aforementioned issues related to the ever-growing pool of sporadic flaky tests.

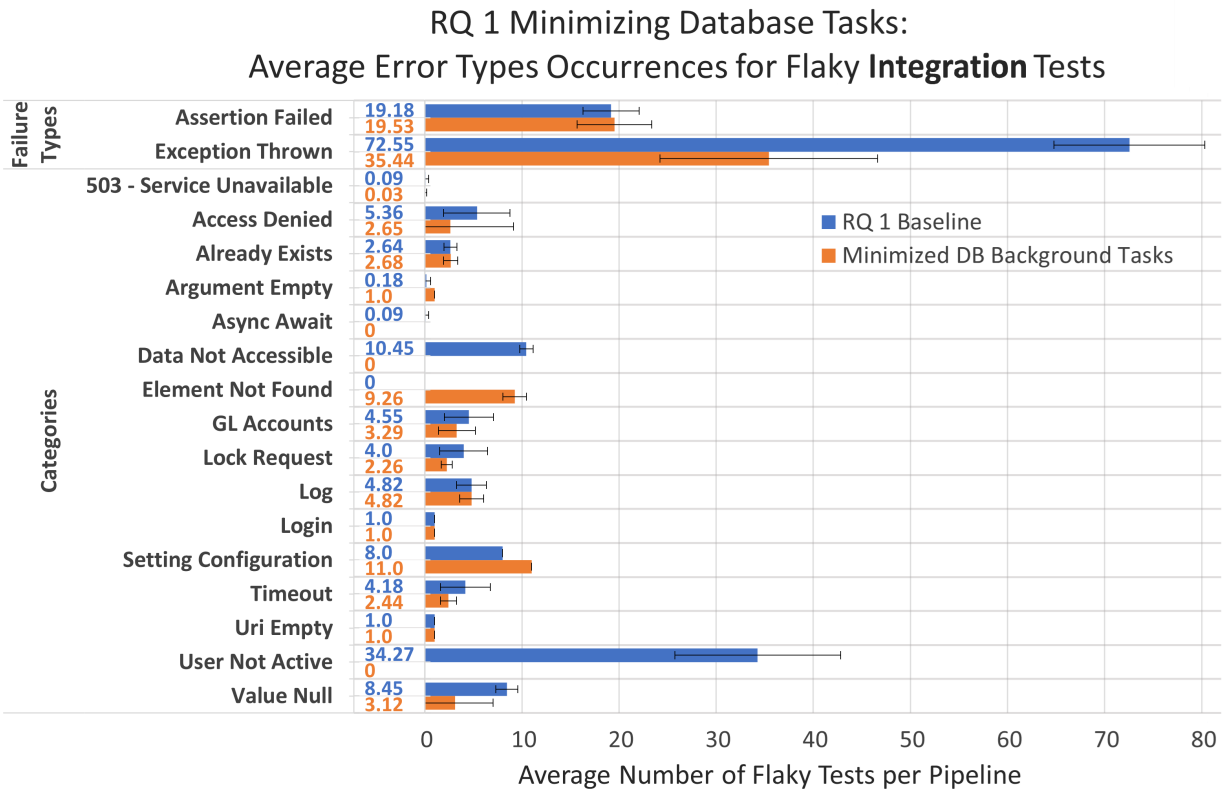
Note that although non-sporadic flakiness is less susceptible to observational randomness, it can still be influenced by it. We observed shifts of more than one pass rate interval attributable to observational randomness. Disabling dirty-row-count tests in Figure 4.12b moves 4 tests to the (0.8, 0.9] interval. Disabling dirty-setting tests in Figure 4.12a moves 14 tests to this interval, with 4 being new compared to the Baseline. However, the third experiment with both groups disabled in Figure 4.12c does not replicate this behavior. Manual inspection confirmed that these significant shifts in pass rates are not indicative of underlying causes but rather artifacts of measurement randomness. All Sankey graph flow changes have therefore been annotated to indicate their probable cause, determined through manual inspection.

4.2 RQ1 - Redundant DB Background Tasks

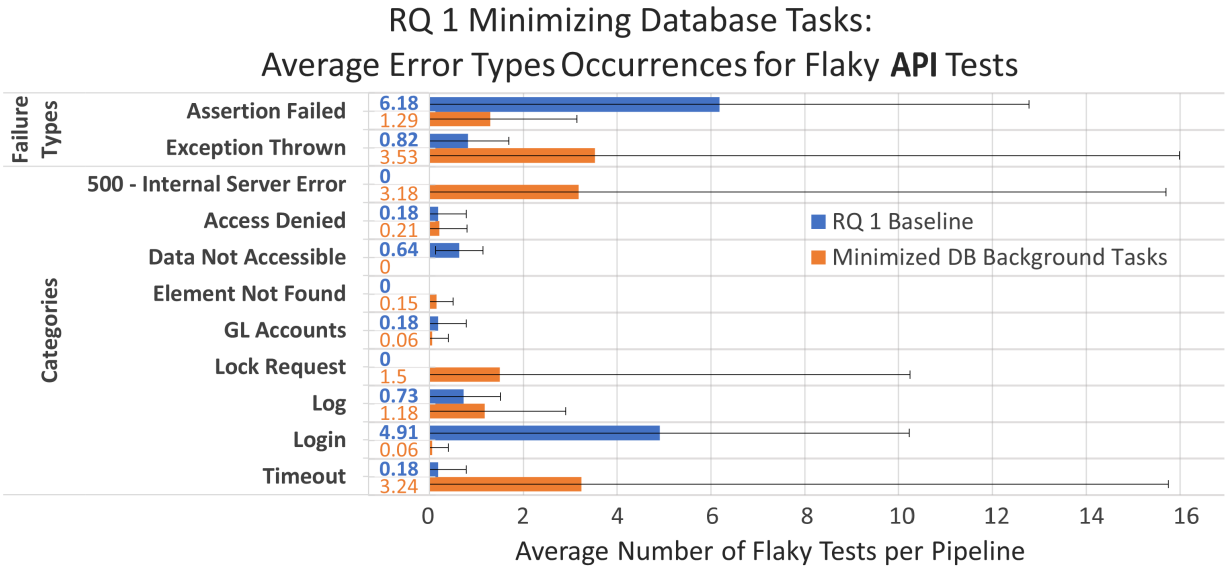
Minimization of the DB background tasks resulted in increased FPPP and a complete reduction in the occurrence of specific test-error types. This section discusses the impact with various levels of granularity starting with the overall impact and the associated flaky categories. It then discusses outliers, oddities and their reasons. Finally, at the highest granular level, we consider change in flakiness for individual tests. In combination with manual inspections we discuss which flakiness is solved but also which new type of flakiness are seen and why. Finally, after providing various granular levels of information, this section conclusion with the effectiveness of this approach, its down-side, and the impact.

4.2.1 Overall Impact

The approach outlined in Section 3.2.1, which involves minimizing all redundant background tasks, resulted in a 1.6-fold increase in FPPP and a 40% reduction in the average number of flaky tests exhibiting. At the outset of this thesis, Exact had a combined FPPP of 27%, comprising 36% FPPP for Integration tests and 64% FPPP for API tests. Following the minimization of DB background tasks, we observed an increase in the combined FPPP to 44%, with 59% FPPP for integration tests and 76% FPPP for API tests. These results indicate significant improvements, which can be attributed to specific error-type categories.



(a) Integration tests



(b) API tests

Figure 4.3: The average distribution of flaky tests observed within RQ 1 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. All test failures are categorized as 'Assertion Failed', 'Exception Thrown', or both. The provided error ranges, based on one standard deviation, only indicate the variability within the sample data, as the samples are not statistically independent.

All tests within the most common error-type categories were resolved by minimizing DB background tasks. Figure 4.3 illustrates the observed average number of flaky tests per pipeline run for each identified category. Note that these categories are overlapping, meaning that a single test attempt failure can be classified under multiple categories. In the baseline run, the most prevalent category for both API and Integration tests was 'User Not Active', occurring almost five times more frequently than the second most common category, 'Setting Configuration'. The 'User Not Active' category accounted for approximately 35% of all test attempt failures and 37.5% of all pipeline failures. After minimizing DB background tasks, this category was completely eliminated.

We observed similar behavior within API tests for its most common category, 'Login', where minimizing background tasks resulted in a reduction of approximately 98.7%. This category was responsible for 75% of failed API stages and 37.5% of all failed pipelines. Collectively, these two categories, due to their overlap, caused 62.5% of pipeline failures. Therefore, resolving these two categories addressed the majority of pipeline failures.

Furthermore, we observed substantial improvements for less prominent error-type categories. As shown in Figure 4.3a, categories such as 'Access Denied', 'Timeout', and 'Value Null', which typically caused failures for 4-9 tests per pipeline, now typically affected only 2 tests. These improvements collectively resulted in a 40% decrease in the average number of flaky tests exhibiting per pipeline, as evident in Figure 4.4.

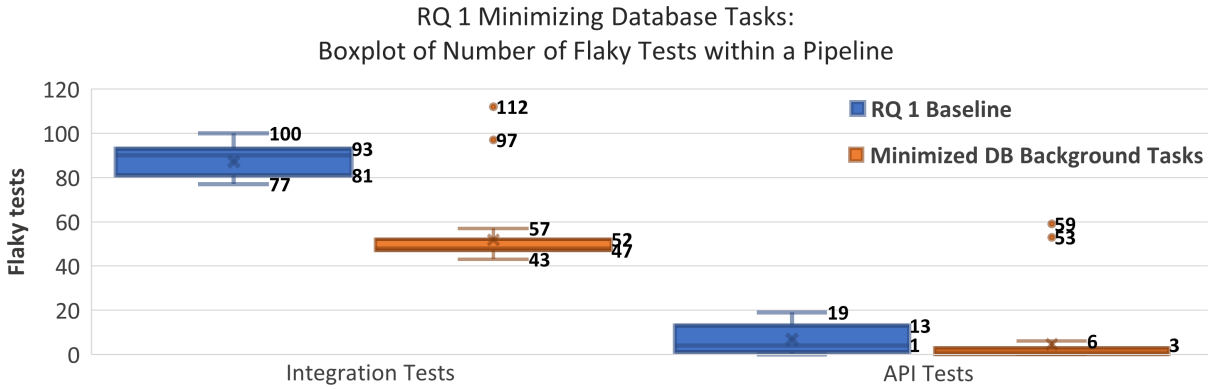
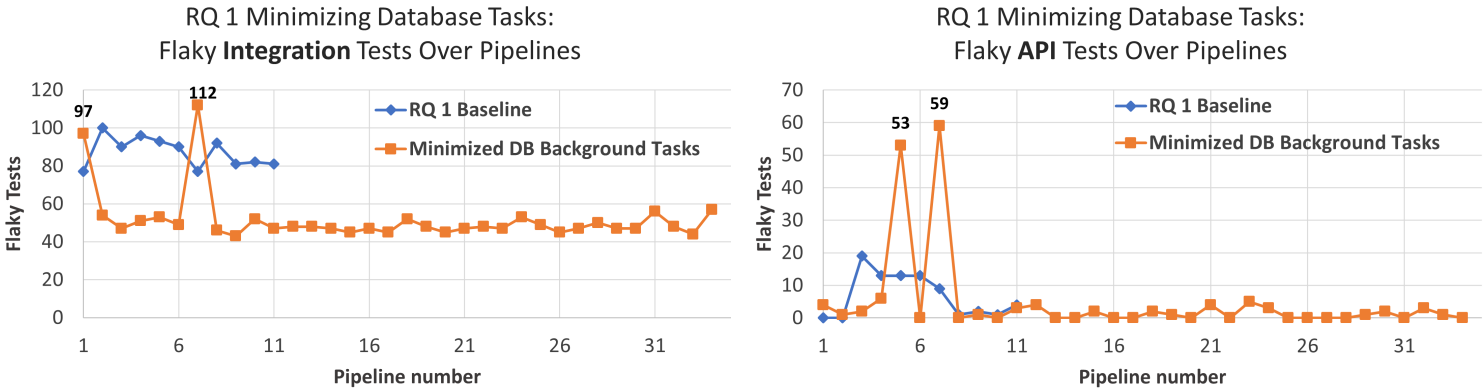


Figure 4.4: Distribution of number of flaky tests within a pipeline, before and after minimizing db background tasks.

4.2.2 Outliers

After minimizing DB background tasks, certain categories appeared to have become more frequent, but this is likely due to observational randomness and external factors. These categories are '500 - Internal Server Error', 'Lock Request', and 'Timeout' within API tests following DB background task minimization. Figure 4.4 shows not only that minimizing DB background tasks resulted in significant improvements but also that the 'Minimized DB Background Tasks' benchmark contained two pipeline runs with an unusually high number of flaky tests in both API and Integration tests. This is indicated by the runs with 53 and 59 flaky tests in API test and the 97 and 112 in Integration tests, instead of their respective median number of flaky tests, 1 and 42. These outliers are also clearly visible in Figure 4.5.



(a) Integration tests

(b) API tests

Figure 4.5: Flaky test counts over pipeline runs for RQ 1 Benchmarks.

Through manual inspection of these four pipeline runs with an unusually high number of flaky tests, we found that for API tests, most failed attempts originate from the same type of issues. Within the two outlying API pipeline runs, 118 test attempt failures were observed across 3 assemblies, all failing with 1 of 8 distinct error messages. The failed test attempts were further automatically grouped by failure types and categories deduced from their error messages. These groups, along with the number of failed attempts, are as follow:

1. Only 'Assertion Failed': 1 failed test attempt
2. 'Assertion Failed' and 'Log': 12 failed test attempts
3. 'Exception Thrown', 'Timeout' and '500 - Internal Server Error': 54 failed attempts
4. 'Exception Thrown', 'Timeout', 'Lock Request' and '500 - Internal Server Error': 51 failed attempts

This analysis reveals that 105 failed test attempts belonging to groups 3 and 4 are highly likely to be related. Due to the cryptic nature of API error messages, it is difficult to definitively infer the exact cause of the issue. However, in this instance, we assume that these groups share a common underlying cause due to their high degree of overlapping categories. The two most frequent and highly similar categories, groups 3 and 4, originate from the same assembly. Manual examination reveals that groups 3 and 4 appear to indicate a timeout during their API request, while group 2 seems to suggest a lack of records. We assume that these timeout issues are indicative of external factors influencing flakiness, similar to those described in Section 4.1.2.

The 12 failed attempts in group 2 are attributed to only 6 tests, and 1 of these tests appears to have become consistently flakier after minimization. The reason for this specific test's increased flakiness is discussed in Section 4.2.3. Group 1, only 'Assertion Failed', is more difficult to categorize, as this can arise from various sources. However, given its small size, we can consider it to be expected flakiness and not necessarily stemming from the same underlying cause as the outlier.

Within Integration tests, the pipeline runs with an unusually high number of flaky tests can be associated with the failure type 'Exception Thrown' and four categories: 'Access Denied', 'Element Not Found', 'GL Account', and 'Value Null'. In the overall flakiness analysis, these

categories exhibited a reduction in frequency relative to the baseline (Figure 4.3). Within the two outlier pipelines, these categories exhibited 40, 7, 11, and 33 more tests than their respective 75th percentile values. This suggests that the actual reduction might be greater than visualized in Figure 4.3 due to the skewing effect of the outliers. The pipeline runs with an unusually high number of Integration flaky tests can also be traced back to a single assembly. Among the 18 assemblies causing 274 failed Integration test attempts in the two pipelines, one assembly stands out by causing 125 failed test attempts. Of these, 110 belong to one of the four previously mentioned categories, while the remaining 15 are classified solely under 'Assertion Failed'. To further emphasize the unusual nature of these failures, this assembly did not exhibit any failed attempts in the other 32 pipeline runs.

4.2.3 Individual Improvements

Minimizing DB background tasks resolved approximately half of all *non-sporadic flaky tests*. Figure 4.6 shows that the baseline benchmark includes 115 tests with pass rates lower than 0.9 (non-sporadic flaky tests), whereas the 'Minimized DB Background Tasks' benchmark only contains 56 non-sporadic flaky tests. This change was achieved by resolving 60 tests with pass rates lower than 0.9, which consistently began to pass, indicated by the green flows annotated with a '+'. However, there are also 10 new tests that previously always passed and now exhibit non-sporadic flakiness, indicated by the red and yellow flows annotated with a '-'. We manually inspected these tests and discuss the changes in the following subsections.

60 Improved Tests

Most of the 60 tests that began to pass consistently belong to one or more of the following categories 'Access Denied', 'Login', 'Timeout', 'User Not Active', or 'Value Null'. These categories also exhibited the most substantial improvements, as seen in Figure 4.3 and Section 4.2.1. Of these 60 tests, 63% are related to 'User Not Active'. This 63% primarily resided in tests with pass rates ≤ 0.6. The other categories observed within these 60 tests included 'Element Not Found', 'GL Accounts', 'Lock Request', and 'Log'. Three of these 60 tests did not fall into any specific category other than 'Assertion Failed'. Notably, these 60 tests originated from only 6 assemblies.

10 New Flaky Tests

The 10 new non-sporadic flaky tests appear to be attributable to dirty tests. They do not share any obvious commonalities. The only discernible pattern is that most seem to indicate missing data in the database. For example, one test falls under the failure type 'Assertion Failed' and category 'Log', indicating missing records, while three others indicate missing configurations for certain settings.

Three of these 10 new flaky tests may not be directly related to missing data. Their failure reasons are more cryptic, such as a simple 'Assert.IsTrue failed' or a timeout. Importantly, **none of these 10 new errors seems to indicate an expected dependency on the background tasks that were minimized**. For instance, a failure would not be expected if the database contained more data than anticipated.

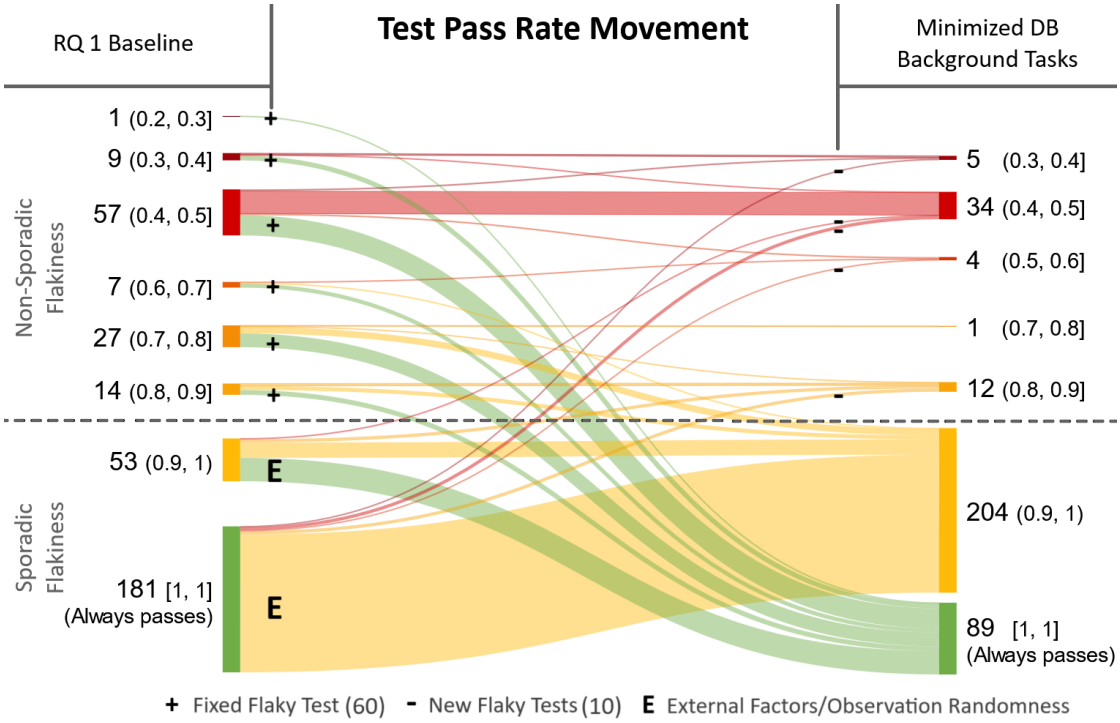


Figure 4.6: Change in pass rates for individual tests after minimizing DB background tasks.

41 Unaffected Flaky Tests

For the 41 non-sporadic flaky tests that remained flaky, no single category encompasses all instances. However, the error messages remain largely consistent across all failed attempts for each test. Eight of these tests failed due to 'Setting Configuration', nine due to 'Element Not Found', and several due to 'Access Denied', 'Already Exists', 'GL Accounts', or 'Log'.

An interesting observation is that 30 of these 41 tests originate from only 3 assemblies, while the remaining 11 are distributed across 8 assemblies. This concentration of flaky tests within a few assemblies could indicate error propagation within these assemblies or test dirtiness if the errors consistently occur on the first attempt. Since the test execution order within an assembly is consistent, if another test within the same assembly were to corrupt data, it would impact subsequent tests within that assembly on every pipeline run. This is precisely the behavior observed, with 28 of the 30 tests exhibiting pass rates below 0.5. The remaining two tests exhibit different error types. Notably, the three assemblies containing these 30 tests are among the 13, 15, and 74 largest assemblies in terms of the number of active tests.

Other Movements

The shifts observed between neighboring intervals are attributed to observational randomness, as explained in Section 4.1.5. The magnitude of the shifts between the (0.9, 1) and [1, 1] intervals can primarily be attributed to sporadic flakiness that due to its sporadic flakiness is unlikely to occur in both benchmarks, as is discussed in Section 4.2.2.

4.2.4 Conclusion RQ1: Minimize DB Background Tasks

By minimizing database background tasks, we observed significant improvements within a database-heavy system such as Exact. Approximately half of all flaky tests with pass rates lower than 0.9 were resolved. Most of these flaky tests were attributed to a single category based on similarities in their error messages. This category was completely resolved.

We observed the emergence of some new non-sporadic flaky tests following the minimization of DB background tasks. However, none of these new flaky tests exhibited an expected dependency on the background tasks that were minimized.

The direct impact of minimizing DB background tasks resulted in a 1.6-fold increase in FPPP and a 40% reduction in the average number of flaky tests exhibiting. This translates to a reduction in the computational resources consumed by rerunning flaky tests and a decrease in the wasted time of developers investigating false test failures caused by flakiness. Consequently, these improvements contribute to lower costs and increased development efficiency.

Minimizing DB Background Tasks: The low-hanging fruit to prevent flakiness

RQ1 Answer: Minimizing DB background tasks reduced the average number of flaky tests exhibiting by 40% and completely resolved 53% of the non-sporadic flaky tests at Exact. Ultimately, this resulted in a 1.6-fold increase in FPPP. No direct causally observed downsides were identified.
Implication: Database-heavy systems that experience flakiness in database-related tests should re-evaluate the necessity of each database background task.

4.3 RQ2 - Implicit Disposing

For this study, we developed and employed a tool called 'WeDispose' to identify and explicitly dispose of all instances of implicit disposing within tests, as outlined in Section 3.2.2. This resulted in fixes for several seemingly unrelated tests. 'WeDispose' successfully identified and refactored over 5,064 instances of implicit disposing, indicating its presence in well over 10% of API and Integration tests.

The following sections, present the overall changes and impact of explicitly disposing test data and detail the behavior of individual tests to understand how explicit disposing can impact test executions and flakiness.

4.3.1 Overall Impact

The 'WeDispose' tool successfully identified that $>$ 10% of tests implicitly disposed of data. Moreover, we found 153; 2,087; and 2,784 missing disposes in methods attributed with `ClassCleanup`, `TestCleanup`, and `TestMethod`, respectively. After explicitly disposing of all these instances, there were only minor improvements in overall flakiness at Exact. With only 1 failed pipeline out of 26 in the baseline run for RQ2, the initial FPPP was 96%. The FPPP after explicit disposing, with 7 failed pipelines, decreased to 73%. We attribute this lower FPPP to observational randomness and the effects of external factors, as we did not observe any substantial changes

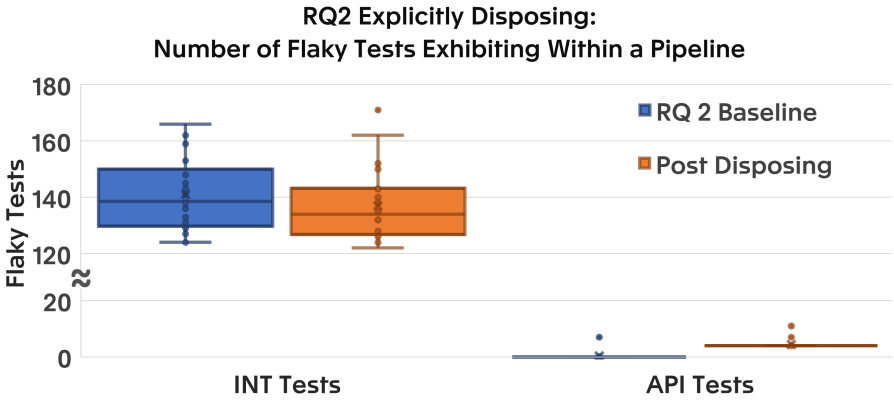


Figure 4.7: Distribution of number of flaky tests within a pipeline before and after explicitly disposing test data.

in overall pass rates, even for the tests responsible for the pipeline failures. Note that the initial FPPP was already relatively high due to manual interventions by Exact developers (Section 4.5).

There are two very similar tests responsible for the seven pipeline failures observed in the WeDispose experiment. These two tests exhibited a pass rate of 0.5 in the baseline and a pass rate of 0.48 after explicit disposing. The single test that caused the baseline to fail also exhibited flakiness in both experiments.

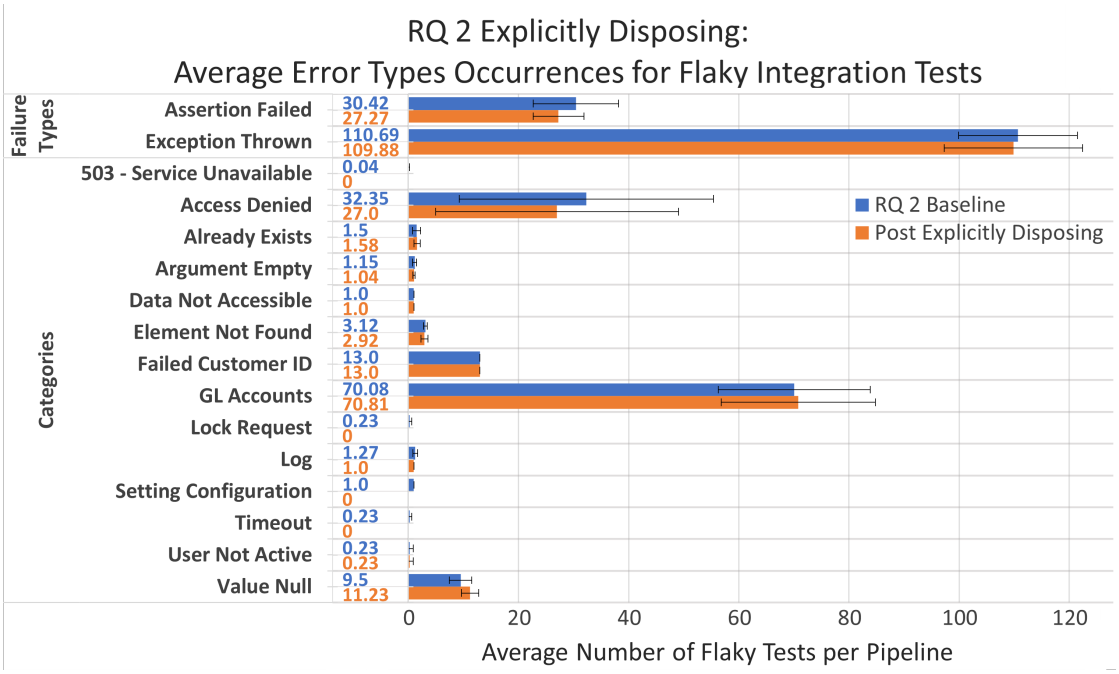
Figure 4.7 shows that, in terms of median instances, there are on average four fewer flaky tests within integration tests and four more flaky tests within API tests for the 'Post Disposing' benchmark compared to its baseline. This observation is further supported by the combined amount of failure types in Figure 4.8. Manual inspection reveals that the four newly flaky API tests consistently failed on their first attempts. All four of these belong to the same class, produce the same error message, and have not undergone any code changes.

Further analysis reveals that the error arose because another test class within the same assembly had undergone code changes. This modified test class, which is responsible for the test failures, disposes of an object retrieved through a factory class. This disposal unintentionally disposes of data set within a method from another test class that utilizes an 'AssemblyInitialize' attribute, [1] which causes a method to be called once at assembly initialization, preceding any class-level initialization.

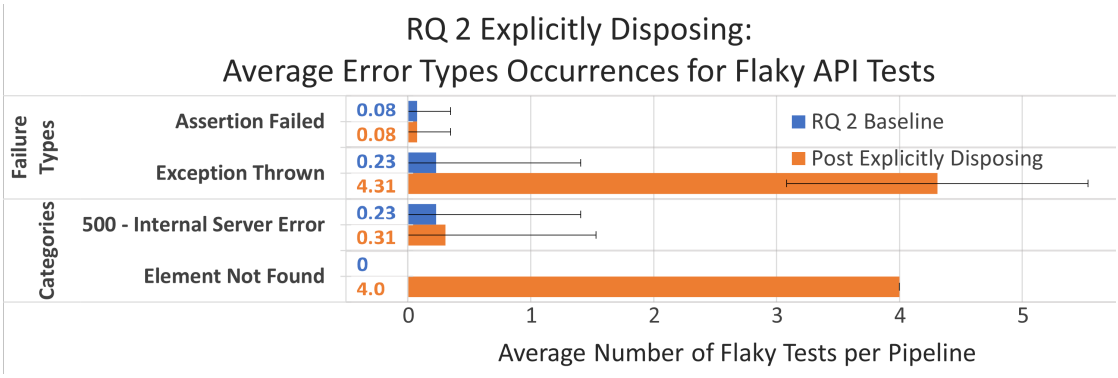
This behavior demonstrates that explicit disposing can, in itself, introduce a type of flakiness that it aims to prevent: Data manipulation in one test class can affect the outcome of another test class. However, this phenomenon is not frequent and might indicate a poorly implemented factory method or excessively high coupling between tests.

Figure 4.8a shows that there are some improvements within integration error types; however, these changes fall within the standard error ranges for each category, highlighted by the black ranges. The standard error is a statistical measure that quantifies the variability of a sample statistic, assuming statistical independence.

[1] https://learn.microsoft.com/en-us/previous-versions/visualstudio/visual_studio-2008/ms245278(v=vs.90)



(a) Integration tests



(b) API tests

Figure 4.8: The average distribution of flaky tests observed within RQ2 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. All test failures are categorized as 'Assertion Failed', 'Exception Thrown', or both. The provided error ranges, based on one standard deviation, only indicate the variability within the sample data, as the samples are not statistically independent.

While we do not always have statistical independence, as demonstrated in Section 4.1, these error ranges still provide an indication of the variance and the potential number of flaky tests observed per category based on observational randomness with the utilized number of pipelines within our benchmark.

We manually inspected the root cause of these changes within the error types for Integration tests to ensure that they actually stemmed from explicit disposing. This analysis revealed only three causally related changes: a decrease of 0.27 tests on average in the 'Log' category and reductions of 1 and 3 tests on average in the 'Exception Thrown' and 'Assertion Failed' failure types, respectively.

4.3.2 Individual Improvements

This section examines individual changes in pass rates, highlighted in Figure 4.9. This figure depicts that several tests were fixed or became flaky, and a substantial group of tests started to fail consistently. Interestingly, manual inspection indicates that most of these changes did not manifest in tests directly modified by the tool. The following sections detail the behavior of each group and discuss their characteristics.

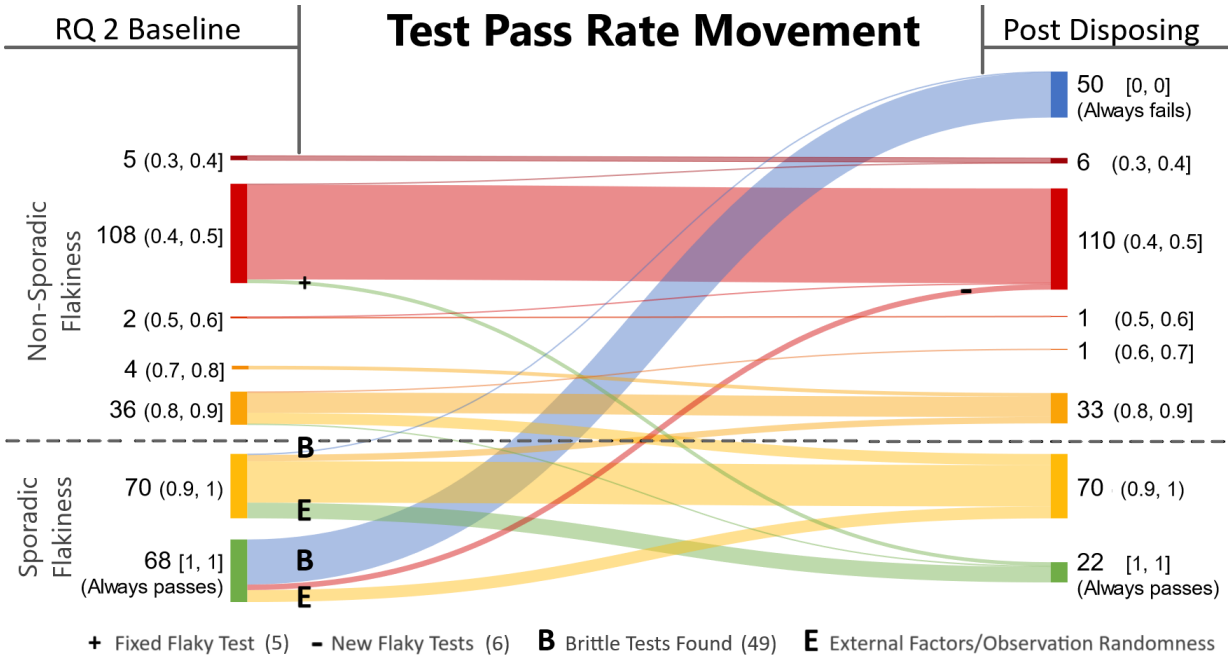


Figure 4.9: Change in pass rates for individual tests after explicitly disposing data.

5 Fixed Flaky Tests

Four of the five fixed tests where always first-attempt-failing tests (i.e. test that consistently failed on their first attempt), as indicated by the green line annotated with a '+' moving from '(0.4, 0.5] RQ2 Baseline' to '[1, 1] Post Disposing' in Figure 4.9. These four tests failed with

assertions regarding unexpected data. The first three of these four tests originate from the same assembly. The fifth fixed test did not consistently fail on its first attempt and exhibited a pass rate of 0.88, as indicated by the thin green line annotated with a '+' moving from '(0.8, 0.9] RQ2 Baseline' to '[1, 1] Post Disposing' in Figure 4.9. This test failed due to a database query timeout and originates from a different assembly than the other four fixed tests. Given the drastic shift from consistently failing on the first attempt to never failing, we can confidently attribute this change in behavior to explicit disposing test data.

Interestingly, all five of these fixed tests do not exhibit code changes within their own class, parent class, or any methods they utilize. This suggests that the fix is not directly related to the code within these tests. The first three tests, which originate from the same assembly, do exhibit file changes within that assembly. The other two tests do not even exhibit changes within their respective assemblies.

As mentioned, the fourth test that consistently failed on its first attempt does not exhibit code changes within its assembly. This implies that the fix originates from another assembly. This is unexpected, since we consider the test order across assemblies random and always first-attempt-failing tests typically fail due to another test which is *always* executed before the always first-attempt-failing test. However, in reality, the randomness is gradual. As explained in Section 2.2, the first tests are distributed across all machines used within the stage, resulting in a consistent order for the first five assemblies. Subsequently, the test order for subsequent assemblies exhibits increasing randomness, as test assemblies are selected from a queue based on alphabetical order. The fourth test, which consistently failed on its first attempt despite code changes within its assembly, is the 11th integration test assembly. This explains why it can consistently fail due to an issue originating from another assembly. If this test were not among the first assemblies, we would expect to observe the same behavior as with the fifth test, where the fix/offending test is located within another assembly, and the pass rate is higher than 0.5 but still substantial ($\leq$ 0.9). This is because the probability of the offending test assembly being executed on the same machine decreases to 1/5 (20%) with Exact's setup of distributing Integration tests across five machines.

Overall, **these fixed tests highlight the challenges associated with identifying the root cause of flakiness introduced by implicit disposing**. The cause can transcend the code directly touched by stack traces for a specific test (potentially residing in databases or singletons) and may not even be located within the same assembly. Furthermore, these examples emphasize the impact of test order randomness on existing flakiness.

6 New Flaky Tests

Six tests consistently failed on their first attempt after explicit disposing, indicated by the red line annotated with '-' moving from '[1, 1] RQ2 Baseline' to '(0.4, 0.5] Post Disposing' in Figure 4.9. These tests originate from two assemblies and include both API and Integration tests. Four of these tests were previously identified in Figure 4.8b and are discussed in Section 4.3.1.

The other two Integration tests originate from one test class and exhibit an error message suggesting a common underlying issue related to a missing setting. These two tests exhibit code changes within their own class. However, these code changes involve cleanup operations performed after each test. These changes should not directly affect the setting within a test itself.

Nevertheless, the consistent failure of these tests indicates that explicit disposing has indirectly introduced this flakiness, either within the same class or elsewhere. This highlights the fragility of the test suite.

50 New Always-Failing Tests

A significant number of tests (50) consistently failed after explicit disposing, indicated by the blue lines moving from '(0.9, 1) RQ2 Baseline' and '[1, 1] RQ2 Baseline' to '[0, 0] Post Disposing' in Figure 4.9. These tests originate from 10 assemblies and exhibit varying error types within each assembly. The four most common error types are as follow:

1. 'Async Await' for 6 tests where only the 'TestCleanup' method has changed
2. 'Value Null' for 7 tests where only the 'TestCleanup' method has changed
3. 'Unexpected Status' for 9 tests where no changes have been made within the assembly
4. 'Key Not Present' for 20 tests where half contain changes only within the test itself or another test in the class, while the other half also exhibit changes in the 'TestCleanup' method.

Group 1 ('Async Await') reveals test dirtiness not only in terms of data but also threads. These tests in group 1 failed due to an 'Async Await' issue, indicating that background threads spawned by the test are still performing operations after the test has completed. These operations can potentially hold locks on the database, leading to subsequent test failures. These tests in group 1 exhibit the 'Fire and Forget'[2] test smell [31], where background threads or tasks are launched without proper management. As discussed in Section 3.1.2, this is one of the two possible primary reasons for database unavailability issues manifested as 'Lock Request' and 'Timeout' errors. As stated in Section 4.2, after minimizing DB background tasks, we observed a partial reduction in these errors, addressing one of the primary causes of these issues. By explicitly disposing of resources, we unintentionally uncovered this 'Fire and Forget' smell, addressing the other primary cause. This corresponds with the observation in Figure 4.8a that after addressing both primary causes by explicitly disposing and minimizing DB background tasks, no 'Lock Request' or 'Timeout' errors are encountered.

Note that while the tests in group 1 are not observed to be flaky, as they always fail, they are still inherently flaky due to the timing-dependent nature of their failures. The group 1 test failures fall under the 'Async Await' flakiness category [64, 23]. This illustrates how addressing one type of flakiness can introduce another, both stemming from the same root cause.

Whether the new 'Async Await' flakiness, with an average fixing effort of 3 out of 5 [23], is more easily addressed than the original flakiness caused by dangling threads is beyond the scope of this research.

Groups 2 ('Value Null') and 4 ('Key Not Present') consist of brittle tests that rely on other tests to establish the necessary data. Since the other tests now perform cleanup operations, the required data is not available, leading to test failures. We consider brittle tests dangerous, as

[2]Fire and Forget smell: if a test launches background threads or tasks.

they obscure the root cause of failures. When the test responsible for initializing the data fails early, it can trigger failures in 10-20 other tests with different error messages. Another scenario highlighting the danger of brittle tests, involves a pull request that includes changes to the test responsible for initializing the data. These changes inadvertently cause failures in brittle tests, leading to confusion and increased development time for determining the root cause of the test failures.

Group 3 ('Unexpected Status') encompasses all tests within a single assembly. This test assembly utilizes SpecFlow and is therefore not directly targeted by our tool. The error message suggests an issue with a misconfigured setting. This observation highlights that SpecFlow-generated tests can also contribute to and suffer from test dirtiness. This finding indicates that incorporating support for SpecFlow-generated tests within the refactoring process could yield further improvements. This aspect is discussed further in Section 5.3.2.

Other. One of the 50 tests that consistently failed after explicit disposing exhibits a failure caused by an incorrect refactoring performed by the tool. Although all refactors were manually inspected to account for edge cases, one mistake slipped through. WeDispose moved a statement from the `try` block to the `catch` block, inadvertently breaking the test. All other refactors are considered correct and showcase how the system is expected to behave.

210 Unaffected Tests

For 210 tests, the pass rate remained unchanged after explicit disposing. Notably, 159 of these tests originate from the 23rd largest test assembly. This disproportionate number of flaky tests within a single assembly suggests a common underlying issue.

Investigation revealed that a singleton object within this assembly injects a stub into another class but fails to perform proper cleanup. This leads to various issues within the assembly, manifesting as different error types. The errors thrown by this bug primarily include 'Access Denied', 'GL Accounts', and 'Value Null'. This observation, combined with the fact that 96 of these tests consistently failed on the first attempt, explains the significant presence of these categories in Figure 4.8a.

For the remaining 51 tests, 12 were already identified as flaky in the RQ1 baseline. Of these 12, 4 exhibit 'Access Denied' errors, while the other 8 exhibit only 'Assertion Failed' errors. For the other 39 tests, 14 exhibit only 'Assertion Failed' errors. The final group of 6 tests, exhibiting a pass rate of only 0.33, failed due to 'Failed Customer ID' errors.

4.3.3 Conclusion RQ2: Disposables Add Complexity

Explicitly disposing of resources is considered good practice and can prevent certain rare but potentially severe bugs. In this research, we developed 'WeDispose' a C# and VB code rule to explicitly dispose of all undisposed objects within all non-generated Integration and API tests at Exact. This process resulted in 2,415 test file changes, with 2,414 being correct, according to Exact design guidelines.

Despite the significant scope of this refactoring effort, we observed minor improvements in overall flakiness at Exact. While 11 flaky tests caused by implicit disposal were already successfully resolved before the baseline benchmark, we still observed a clear change in behavior for

61 tests. The tool successfully resolved 5 flaky tests by explicitly disposing of test data within tests in other classes or assemblies. However, this correct refactoring also resulted in 6 new flaky tests. All of which consistently failed on their first attempt due to changes in other tests, highlighting how test suites can become brittle as a result of high levels of coupling.

Furthermore, explicit disposing resulted in 50 tests to consistently fail. Forty-nine of these failures are not attributable to incorrect refactoring performed by the tool but rather reveal underlying issues within the test suite. These issues include the following:

- **Dangling Threads:** The 'Fire and Forget' test smell, where background threads or tasks are launched without proper management, was identified in a group of 6 tests. This issue contributes to database unavailability and subsequent test failures.
- **Brittle Tests:** Twenty-seven tests exhibited increased flakiness, or started to fail consistently, due to their reliance on other tests to establish the necessary data. When the tests responsible for initializing this data perform cleanup operations, the required data becomes unavailable, leading to test failures.
- **Unexpected Coupling:** In some cases, explicit disposing in one class inadvertently affected the behavior of other classes, leading to unexpected test failures.

These findings demonstrate that explicit disposing can have unintended consequences and may introduce new types of flakiness into the test suite.

Showing, that while the Dispose design pattern is a valuable practice, one must be aware of the *Implicit Dispose* smell. Relying on implicit disposal inadvertently introduces complexity through shared data and potential unintended side effects. Our research highlights the importance of explicitly disposing of resources within testing and the prevalence of Implicit Dispose smell in an industrial setting. It also emphasizes the need for robust test suites with minimal dependencies between tests to minimize the impact of such changes.

The Implicit Dispose Smell Introduces Difficult Bugs

RQ2 Answer: We found 5,064 implicit disposals in tests at Exact prevalent in well over 2415 (10%) of tests. Explicitly disposing all these undisposed instances changed the test outcome for 61 tests, fixing flaky tests and revealing dirty tests, dangling threads, brittle tests, and unexpected coupling between tests. This shows that the highly frequent Implicit Dispose smell in tests cause difficult but very rare bugs, resulting in flakiness.
Implication: Always explicitly dispose of disposable objects to prevent (unintended) coupling and flakiness.

4.4 RQ3 - Dirty Databases

For this study, we developed and employed two database sanity check methods to identify all dirty tests, as outlined in Section 3.2.3. We then subsequently disabled all dirty tests and ran four benchmarks to determine the effect of database test dirtiness on flakiness at Exact. We found that 11.43% of evaluated tests exhibited dirty database behavior, resulting in flakiness for 28% of non-sporadic flaky tests.

The following sections discuss both sanity checks and illustrate an upper bound on their combined overhead. They also detail the behavior of individual tests after disabling these dirty tests to understand how database test dirtiness can impact test flakiness.

4.4.1 Comparing Sanity Checks

Performing sanity checks identified 2,178 tests that modify the database. This represents 11.43% of all tested tests. The generalized row-counts sanity check approach flagged 9.8 times more tests than the specialized setting check. These numbers are visualized in Figure 4.10. Notably, approximately half of all tests flagged by the specialized dirty setting level approach were also flagged by the row-counts check.

Interestingly, although the row-counts sanity check flagged almost 9.8 times more tests than the specialized approach, disabling dirty-row-count tests fixed only approximately $\frac{3}{8}$ 2.7 times as many non-sporadically flaky tests as disabling dirty-setting tests. This observation suggests that dirty tests identified by the specialized approach are more likely to cause flakiness than those identified by the generalized approach.

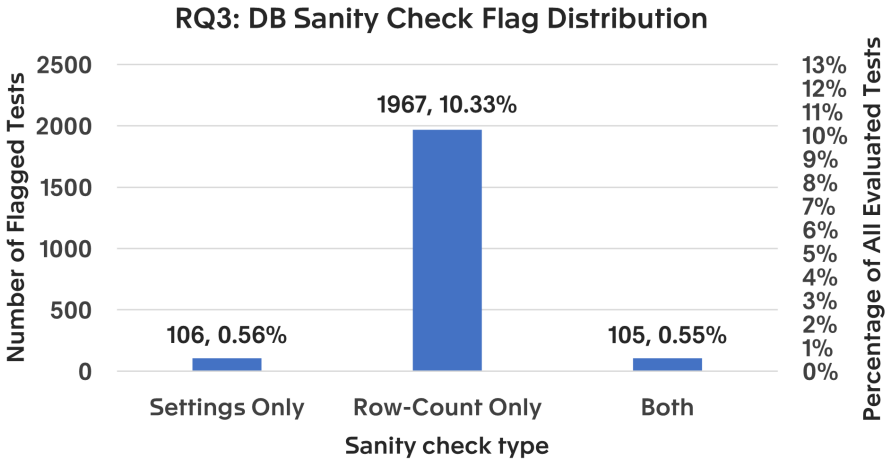


Figure 4.10: Where and how often the DB sanity check flagged tests.

4.4.2 Overhead of Sanity Checks

Sanity checks introduce overhead. Performing both unoptimized DB sanity checks added approximately half a second in the median case. This overhead significantly increased test execution times, shifting the median and mean execution times from 0.140s to 0.671s and 0.492s to 2.353s, respectively, as illustrated in Figure 4.11. This implementation was intended as a proof of concept to assess the impact of sanity checks. Therefore, it should be considered a worst-case scenario in terms of execution time.

Several optimizations can be implemented to improve the performance of the sanity checks. For example, both the pre- and post-DB query currently establish and close a new database connection. Another potential optimization involves the element-wise comparison of every element

in the requested state, regardless of whether the total values differ. In the case of Exact, for tests that previously executed in 20ms, the sanity checks now require establishing two database connections and comparing 1,500 row counts and 11,000 settings.

The key takeaway is that sanity checks introduce additional execution time. The impact of this overhead depends on the current test execution times due to the overhead being dependent on the database size and connection time. If you have a relatively small number of tests with long execution times, incorporating sanity checks into your CI pipeline may be feasible. Alternatively, if you have faster test execution times, performing sanity checks periodically may be a more practical approach to identify dirty tests.

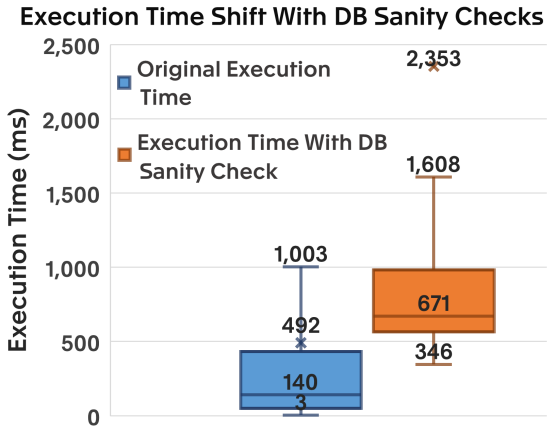
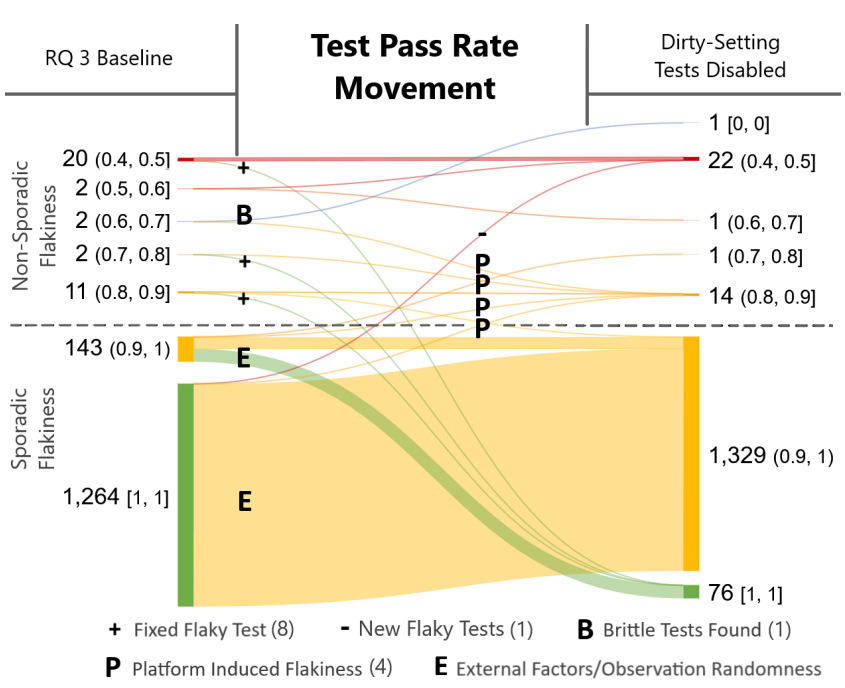


Figure 4.11: Distribution of test execution times with and without sanity checks.

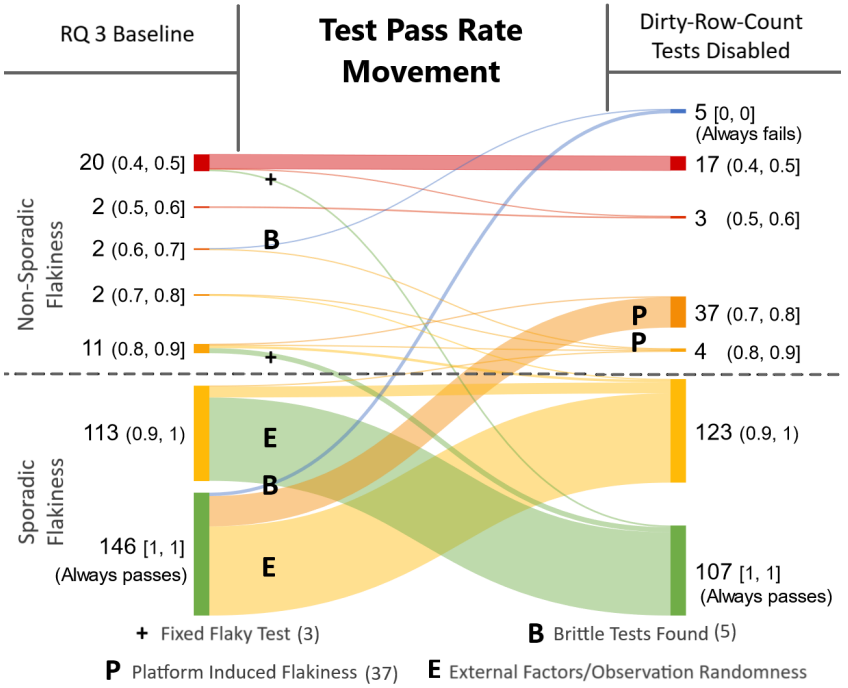
4.4.3 Individual Improvements

This section examines individual changes in pass rates highlighted in Figure 4.12. This figure reveals that disabling certain dirty tests resulted in 5 tests to start consistently failing, and fixing the flakiness of 11 non-sporadic flaky tests. Additionally, it identifies a new group of 2 tests that become non-sporadically flaky.

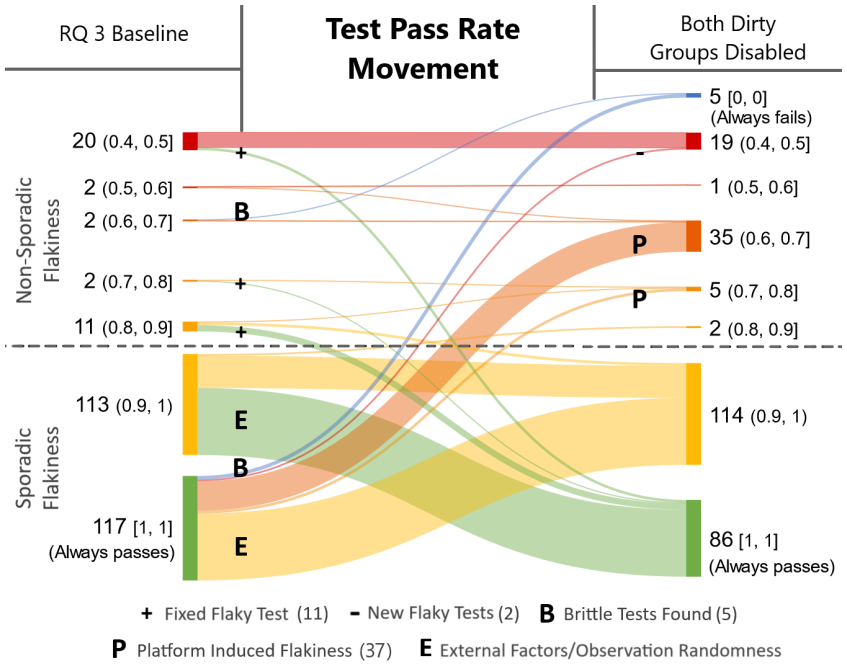
We manually investigated all tests with significant changes in pass rates (Section 4.1.5) and found that these changes stem from a combination of dirty tests, brittle tests, environmental changes, and other side effects. The following subsections present the behavior of each group and discuss their characteristics.



(a) Dirty-Setting Tests Disabled Benchmark



(b) Dirty-Row-Count Tests Disabled Benchmark



(c) Both Dirty Groups Disabled Benchmark

Figure 4.12: Change in pass rates for individual tests between RQ3 experiments.

11 Fixed Flaky Tests

Eleven non-sporadic flaky tests began to pass consistently after disabling both types of dirty tests, as visualized by the green lines annotated with '+' moving from the baseline intervals (0, 0.9] to [1, 1] in Figure 4.12c. These tests exhibit diversity; they belong to both API and Integration test stages, they are maintained by different departments, and they reside in various assemblies.

The fixes for these flaky tests can be attributed to the disabling of either dirty-row-count tests or dirty-setting tests. This resulted in the resolution of 3 and 8 non-sporadic flaky tests, respectively. This is visualized by the green lines annotated with '+' moving from the baseline intervals (0, 0.9] to [1, 1] in their respective graphs, Figure 4.12a and Figure 4.12b.

The consistent resolution of non-sporadic flaky tests in the benchmark with both dirty test groups disabled and the benchmark with only dirty test groups disabled indicates that these 11 tests were likely affected by dirty tests within the system.

5 DB-Brittle Test Found

Five always-passing tests began to consistently fail after disabling both types of dirty tests, as visualized by the blue lines annotated with 'B' moving from the baseline intervals [1, 1] to [0, 0] in Figure 4.12c. Manual verification revealed that these tests appear to rely on specific dirty states within the system.

By combining information from all three benchmarks with different groups of dirty tests disabled, we can link the flakiness to both row-count and setting dirtiness, as well as the location of the dirty state setter test.

The four tests that transitioned from [1, 1] likely rely on row count dirtiness within the same class. This assumption is supported by the observation that the same behavior was observed when only row-count dirty tests were disabled (Figure 4.12b) but not when only setting dirty tests were disabled (Figure 4.12a). The consistent passing behavior of these tests prior to disabling dirty tests, while being dependent on other tests to execute first, suggests they were likely executed in a fixed order. Within Exact, this most likely implies that these tests reside within the same class. This finding highlights that **consistent passing behavior does not necessarily imply the correctness of a test**. Incorrect behavior may emerge later in development, as observed in this case with post-dirty-test-disablement.

The remaining test, which transitioned from (0.6, 0.7], consistently failed in all three experiments. This suggests that it exhibited test-order dependency and was fixed by disabling either type of dirty test. It is likely that this test relied on a test from another assembly (due to test-order dependency) to add a setting, as indicated by the impact of both row-count and setting sanity checks.

2 Non-DB-Data Brittle Tests Found

Two always-passing tests began to exhibit non-sporadic flakiness after disabling both types of dirty tests, as visualized by the red lines annotated with '-' moving from the baseline intervals [1, 1] to (0.4, 0.5] in Figure 4.12a and Figure 4.12c. Unlike the other five brittle tests, these two tests do not rely on database dirtiness but rather on some other type of order dependency.

While the true nature of these failures remains uncertain, the most likely explanation is that these tests were indirectly affected by disabling other tests. This likely involves test-order dependencies, where the new consistent test order, introduced by disabling dirty tests, causes these tests to fail consistently on their first attempt. This test-order dependency is unrelated to database dirtiness checked by the sanity checks, as these tests would have consistently failed if they relied on the database dirtiness, similar to the other five brittle tests.

These tests demonstrate how changes in the test environment, even those intended to improve stability, can reveal hidden brittleness and expose new areas of flakiness. Furthermore, this observation highlights the existence of multiple types of dirtiness or test-order-dependent issues within the test suite, emphasizing that there is no single solution to address all forms of flakiness.

Other Movements

Approximately 37 tests exhibited increased attempt failures, resulting in pass rates within the (0.6, 0.8] interval in the 'Dirty-Row-Count Tests Disabled' (Figure 4.12b) and 'Both Dirty Groups Disabled' (Figure 4.12c) benchmarks. These tests belong to the 'Specflow Assembly Init' category and their increased flakiness is attributed to changes in the test platform, as discussed further in Section 4.1.3. The same tests transition to the (0.8, 1] interval in the 'Dirty-Setting Tests Disabled' benchmark (Figure 4.12a).

The 'Dirty-Row-Count Tests Disabled' and 'Dirty-Setting Tests Disabled' benchmarks exhibit 4 and 14 tests, respectively, within the (0.8, 0.9] interval, with some tests shifting more than one interval compared to the baseline. However, the third experiment with both groups disabled in Figure 4.12c does not replicate this behavior. Manual inspection confirmed that these shifts in pass rates are not indicative of underlying causes but rather artifacts of measurement randomness.

A large number of tests exhibit shifts between the pass rate intervals (0.9, 1) and [1, 1] in all three benchmarks (Figure 4.12). These shifts are primarily attributed to observational randomness and external factors, which are discussed in more detail in Section 4.1 and 5.2.3.

4.4.4 Conclusion RQ3: DB Sanity Checking Reveals Dirty Tests

Sanity checking effectively reveals dirty tests that can contribute to flakiness. We evaluated two different sanity checks: a generalized and a specialized approach. These checks collectively identified 11.43% of tests at Exact as exhibiting dirty database behavior. This demonstrates the effectiveness of DB sanity checking as a method for identifying instances of test dirtiness within the test environment. Disabling these dirty tests resolved 11 flaky tests and uncovered 7 brittle tests, 5 of which exhibited dependencies on the database state.

The generalized approach flags any test that modifies the total number of entries in the database. The specialized approach detects changes in the values of global settings, which are considered incorrect behavior at Exact. While the generalized approach identifies a larger number of dirty tests, tests flagged by the specialized approach are more likely to exhibit flakiness. Therefore, we recommend starting with the generalized approach and then transitioning to a more specialized approach if the number of flagged tests unrelated to flakiness becomes excessive.

Although database dirtiness is prevalent at Exact, it does not always directly translate to significant flakiness. Nevertheless, database dirtiness represents a significant source of flakiness within the test suite. In this study, database test dirtiness was responsible for unintended behavior in 16 tests, including 11 flaky tests and 5 brittle tests. This represents a substantial proportion of the 39 non-sporadic flaky tests observed in the baseline benchmark.

DB Sanity Checks Identify Dirty Test Whereof a Small Percentage Causes Flakiness

RQ3 Answer: We found 11.43% of Integration and API tests at Exact that leave the database in a dirty state, resulting in 7 brittle tests that relied on this behavior and 11 (28%[a]) tests exhibiting non-sporadic flakiness due to database dirtiness. This shows that dirty database tests are frequent within Exact and contribute to flakiness.
Implication: For test suites that suffer from dirty tests or operate under the assumption that no dirty tests exist, database sanity checking provides an effective method for validating the cleanliness of the test environment. It is recommended to begin with a generalized approach and then transition to a more specialized approach if the number of flagged tests not related to flakiness becomes excessive.

[a] of non-sporadically flaky tests

4.5 Total Improvement

Exact has gradually mitigated the impact of flakiness within its system since the start of this study. This is evident in Figure 4.13, which demonstrates a significant improvement in the FPPP, which rose from 27% to 96%, and a substantial reduction in the number of flaky tests following the initial reports on June 6. The size of the interquartile range (IQR) inversely correlates with the consistency of flakiness within a pipeline. 'Flaky Tests Showing' represents the number of tests that failed at least once during each pipeline run, while 'Failed Attempts' denotes the total number of failed attempts within a pipeline run. The difference between these two metrics indicates the number of second or third failed attempts. This difference between 'Flaky Tests Showing' and 'Failed Attempts' is intrinsically linked to the overall test *pass rate*.

4.5.1 Timeline of Benchmark Runs and Influencing Events

Flaky test reports were shared with Exact based on benchmark runs conducted on June 6, June 30, and July 14. Following each report, Exact addressed the flaky test with the lowest pass rate or the most prevalent flakiness type.

Between June 30 and July 14, Exact manually resolved 11 instances of flakiness attributed to implicit disposing. This intervention is reflected in the reduced IQR of flaky tests after June 30th in Figure 4.13. Despite this, the number of flaky tests still increased between June 30 and July 14th due to a new group of flaky tests consistently failing on their first attempt.

Between July 14 and October 10, Exact deployed a revised test logging system to address flakiness related to the 'Log' category, the third most common error message category. This

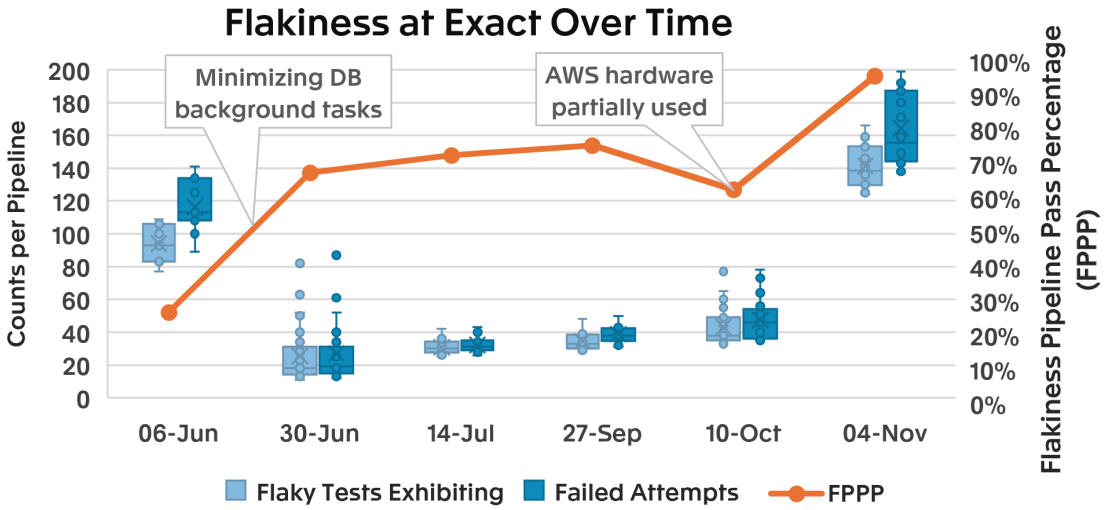


Figure 4.13: Flaky tests at Exact over time, gathered from baseline benchmark runs. Note: FPPP represent how often both the API and Integration test stage passes when reran on the same release with the same configuration.

category was initially associated with an average of six flaky tests per pipeline run, which was reduced to one test per pipeline run following the fix.

The partial implementation of AWS hardware on October 10 resulted in a lower FPPP and a wider distribution of flaky tests. Figure 4.13 illustrates 3 data points that fall outside the 75th percentile of the box plots on October 10. These three data points correspond to 4-7[3] out of 26 runs executed on AWS hardware.

4.5.2 Trends Over Time

A significant decline in flakiness, particularly for tests failing multiple times, was observed following the initial flaky test benchmark. This initial reduction can be attributed to both the minimization of database background tasks and the proactive resolution of specific flakiness categories or tests with the lowest pass rates by developers based on the initial report. The impact of focusing on tests with the lowest pass rates is evident from the new equivalency of the 'Flaky Tests Showing' and 'Failed Attempts' box plots on June 30 in Figure 4.13.

Following the initial report on June 6, the FPPP has consistently increased, with a slight decline observed on October 10 due to the introduction of AWS hardware. The improvement in FPPP can be attributed to the initial improvements and the sustained prioritization of fixing tests that cause RRT failures.

However, **flakiness slowly reenters the system when unmanaged**. While the FPPP has been steadily improving, the total number of flaky tests has also increased since June 30, as visualized in Figure 4.13. This seemingly contradictory trend, similarly observed by other in-

[3]Only 3 out of 7 runs involved executing both API and Integration tests on AWS hardware. For the other 4, only the API test was executed on AWS hardware.

dustrial flakiness studies [55], can, in this case, be explained by the fact that many of these flaky tests pass on subsequent attempts. However, the number of flaky tests that fail on subsequent attempts is also steadily increasing, as evidenced by the growing disparity between 'Flaky Tests Showing' and 'Failed Attempts'. The difference on November 4 is nearly as substantial as that observed on June 6, suggesting a potential imminent decline in FPPP.

4.5.3 Release rate

The release rate at Exact has significantly improved as a result of this research. Successful releases on planned dates are considered crucial at Exact. The release rate metric, which quantifies the frequency of successful releases within a month, is reviewed monthly by Exact's upper technical management. The release rate typically hovered around 80% but it dropped to a record low of 60% in May. Since July, Exact has consistently achieved a record-high release rate of 95% for three consecutive months. The company attributes this improvement, in part, to the flaky test reports obtained from the benchmark runs conducted during this research.

Chapter 5

Discussion

This chapter discusses the key observations and findings made during the course of this study. Section 5.1 explores how flakiness-related information can guide organizational efforts to address flakiness, illustrating both the benefits and drawbacks of specific flakiness metrics and measurement methods. Section 5.2 discusses how flakiness manifested at Exact, the implications of these observations, and strategies for adapting systems or workflows based on specific types of flakiness or combinations thereof. Both sections contain key takeaways highlighted in grey boxes along with their implications for engineers dealing with flakiness. Finally, Section 5.3 addresses potential threats to the validity of this research.

5.1 Rich and Summarized Information Enables Organizations to Combat Flakiness

Our results demonstrate that flakiness can stem from several root causes, rather than being isolated to individual test failures. Therefore, it is crucial to address these root causes as systemic issues. This section discusses how presenting flakiness information in granular, summarized, or grouped views empowers organizations to effectively address flakiness by informing and motivating engineers.

Section 5.1.1 discusses the limitations of existing test result visualizations for identifying overarching root causes of various test failures. We then advocate for the power of multiple same-commit reruns in Section 5.1.2 to generate this data accurately. An approach that enabled us to obtain, correlate, and visualize various factors potentially causally related to flakiness. However, not all the information was deemed equally useful in addressing flakiness and in Section 5.1.3 we discuss this. Interestingly, our findings regarding flakiness-correlating factors, such as test timings, diverged from related work.

During the case study, we observed that providing a summarized view of flaky tests motivated both developers and management to address them. They indicated that quantifying issues and grouping numerous small problems into larger overarching issues facilitated communication about the necessity of investing effort in investigating and implementing fixes. Section 5.1.4 discusses these motivational benefits, while Section 5.1.5 explores the specific effects of our introduced *Flakiness Pipeline Pass Percentage* (FPPP) metric.

5.1.1 Aggregating Test Results Helps Explain Flakiness

Developers often lack contextual information to effectively categorize, group and understand test failures when looking at individual test failure. For instance, our results demonstrate that the assumption of test independence is often invalid in practice, with observed phenomena including brittle tests, cascading failures due to virtual machine (VM) errors, flaky tests caused by database dirtiness, dangling threads, and redundant background processes – findings that are not unique to Exact [10, 37, 46, 56, 92, 110, 119]. These problems transcend individual tests and, as we illustrate by three examples later in this section, when examining these individual test failures it can be unclear what the actual failure cause is.

Furthermore, extensive research on bug reports suggests that grouping them provides a clearer overview of bugs that are difficult to reproduce (e.g., [12, 13, 122]). Consequently, several studies investigate grouping bug reports based on stack traces or report messages [62, 75, 89, 111]). We believe this approach is equally beneficial for flaky tests, as flakiness can be considered a code bug, and the failing test result serves as information reporting on this flakiness bug (i.e., a bug report).

Similarly, analyzing test failures based on test results has also proven useful in the industry, where testers use regular expressions to distinguish test alarms with 20-30% accuracy [48]. Various patents exist for tools that automatically bucket [87] or analyze test failures [49, 103]. Verifying the potential of such tools: **We observed that providing developers with grouped test failures enabled them to deduce underlying root causes more quickly and for more tests simultaneously**, partly supported by the following three examples.

Example 1: Multiple Test Reports Provide Context

The 'Canceled Task' example in Section 4.1.2 involved hundreds of tests failing concurrently with the same cryptic error message and unusual stack trace. Examining a single test in isolation makes it difficult to interpret the error message `... One or more errors occurred ... Aborting test execution`. However, observing a sudden surge of hundreds of tests failing with the same message suggests it is not related to a specific test. While a developer might reach the same conclusion by reviewing individual results, this example highlights how multiple test results can support hypotheses about test failures.

Example 2: Faster Root Cause Deduction

Another instance involved a single failing test causing an RRT (Release Regression Test) failure, prompting investigation and a fix. Simultaneously, another developer was investigating the root cause for a group of flaky tests categorized based on their error message. This second developer not only more easily addressed the test responsible for the RRT failure but also fixed an additional 10 related issues by implementing explicit data disposals. This demonstrates how aggregated information can accelerate the resolution of multiple flakiness problems. This benefit aligns with prior research that found benefits in addressing similar problems within the same context, as developers can avoid the effort of repeatedly locating and understanding the problem [51].

Example 3: Quantifiable Benefit of Aggregated Reports

Our overall results provide the strongest quantifiable argument in favor of aggregated reports. Based on aggregated reports from benchmark runs, we deduced that redundant database background processes were a key factor causing flakiness at Exact (as illustrated in Section 3.1.2). Following this deduction, we were able to reduce the average number of flaky tests exhibiting this behavior by 40% (Section 4.2). Additionally, as shown in Section 4.5, the FPPP improved from 27% to 96%, and Exact achieved record-high release rates since the start of this research, which was fueled by aggregated reports.

Exact's Opinion

While other research explores methods to improve isolation [9] or detect violations [37, 38], we found that supporting developers in identifying non-isolated test faults is equally crucial, especially since in projects like Exact they deem true independence between integration tests inherently impractical due to time constraints. Our reports generated throughout and for this study were well-received by Exact, leading them to investigate the integration of similar regularly generated aggregated flaky-test reports within their CI/CD pipeline.

Aggregating Test Results Helps Explain Flakiness

Analyzing aggregated test results across one or more pipelines provides valuable insights into underlying flakiness issues that may be difficult to identify when examining individual test failures in isolation.
Implication: Providing summarized test result data through dashboards or aggregated reports can significantly improve developer productivity by facilitating root cause analysis.

5.1.2 Same-Commit Reruns Accurately Generate Flaky Information

Other research has gathered information information of the flaky state of the system through various methods, including relying on developer perceptions of flakiness or utilizing CI data over time (Section 2.1). However, these methods do not provide as accurate information as generating data by rerunning the same commit [78]. In this section we illustrate why it is good approach to gather flakiness information by rerunning on the same commit.

Developers Perception

Before commencing this thesis, Exact relied on developer perceptions to tackle flakiness. This approach was inherently biased. For example, they initially believed that UI tests were responsible for most pipeline failures, an assumption seemingly validated by the frequent need to rerun these tests. However, as illustrated in Section 4.5 and other work [55], there is no direct correlation between the number of flaky tests and the number of failed builds, which is Exact's primary concern.

By generating same-commit reruns in conjunction with the FPPP score, we identified API and Integration tests as the most common root causes of pipeline failures. This, along with other reasons explained in Section 3.1.1, led us to focus solely on API and Integration tests in our study, where we observed a substantial increase in the release rate at Exact, from 60% to 95% (Section 4.5.3). This improvement can be attributed in part to a corresponding increase in the FPPP of API and Integration tests from 27% to 96% (Section 4.5.3). Thereby indicating that API and Integration tests are indeed more problematic than UI tests at Exact and thus demonstrating that our rerun approach provides more accurate information than relying solely on developer perceptions.

Temporal Analysis

Another common approach utilized by the Exact developers is to manually assess test flakiness based on the pass/fail history of individual tests. This approach is inline with other methods that automatically determine if a test is flaky based on its failure history [65, 72, 78]. However, such temporal analysis may not accurately reflect the current flaky state of the system. Not only did developers at Exact encountered difficulties in accurately classifying flakiness using this approach, this method also does not consider changes within the tests themselves. Developers still need to manually verify the persistent flakiness of a test, a notoriously challenging task [55, 57]. Some developers relied on changes in the test code to evaluate if their temporal based assessment of flakiness is still valid. A similar approach to research that focused on test versioning based on the last commit of the test (rather than the last commit in the entire repository) to evaluate the version in which the test is flaky [41]. However, this method in itself is inaccurate given that flakiness can also originate from production code changes that are not directly reflected in the test code [23]. Additionally, other studies have observed that the flaky state of a test can change even without any modifications to the code touched by its execution stack [94, 100]. Our findings corroborate this observation, demonstrating that flakiness can arise from:

- Configured background tasks that are not part of the test code (Section 4.2)
- Changes in other test classes and even other test assemblies due to persistent data or platform-related factors (Section 4.3 and 4.4).

Single Observations

Multiple runs are necessary for accurate flakiness estimation, as a single pipeline run can be susceptible to external factors that induce flakiness. As described in Section 4.1, we differentiated between *non-sporadic* and *sporadic* flakiness. This distinction is crucial because certain tests may occasionally fail due to external factors or test-order dependencies. These flaky test failures due to external factors can dilute the set of flaky tests masking more impactful flakiness which leads to pipeline failures. It is essential to differentiate between environmental flakiness and each type of test-specific flakiness, as they exhibit differently and require different resolution methods [23], as discussed in Section 5.2.

Other work has developed various tools to pinpoint flakiness, some of which only need one run. However, as discussed in Section 2.1, tools to detect flakiness often miss flaky tests, produce false positives, or inaccurately represent the actual impact of flakiness within the system [4, 11, 38, 60, 83]. Similarly, our WeDispose and the DB sanity check methods, which targeted specific types of flakiness, identified more violating instances than tests that actually caused flakiness (Section 4.3 and 4.4.1), thereby misrepresenting the amount of actually flaky tests within the system.

In conclusion, for the sake of accuracy do we advocate for the powerful and straightforward approach of rerunning the CI pipeline multiple times on the same commit to generate data that accurately reflects the flakiness of the system during a specific commit.

Same-Commit Reruns Accurately Generate Flaky Information

Analyzing multiple test reruns on the same commit with the same configuration and hardware can accurately determine the flaky state of the system and help differentiate between different types of flakiness and external factors.
Implication: Rerunning tests on the same commit several times can provide strong test flakiness data for research, analysis, tools, and development.

5.1.3 Not All Information Is Equally Useful for Addressing Flakiness

The goal of addressing flaky tests at Exact included finding the root causes of flakiness (RQ0). This involved analyzing and grouping flaky tests based on influencing factors and presenting this data to engineers at Exact. We received informal feedback regarding the usefulness of various information sources used to represent test results, such as test timings, pass rates, exception types, and error types. Some of this information was more representative of flakiness or helped engineers better understand the problem. This section discussed these observations to inform future work and the software industry where to focus on when addressing flaky tests. While these observations are specific to Exact, we believe they may be indicative for many other software projects. Interestingly, some of the findings, such as the relation between flakiness and test timings [4, 63], did not coincide with other work.

Less Representative Metrics for Flakiness at Exact

We investigated the relationship between flakiness and several information sources. In this subsection we discuss these sources that did *not* provide valuable insights for our analysis or for the developers at Exact.

Execution Time: Previous research has suggested a potential connection between test timings and flakiness [4, 63]. We investigated this relationship within the context of Exact's test suite, which exhibited a wide range of execution times: the longest test took over 4000 times longer than the shortest and over 400 times longer than the median. Despite this significant variation, we found a limited correlation between test execution time and flakiness.

However, our analysis revealed that the average execution times of failing attempts differed from passing attempts, with failing attempts taking between 0.005 and 1065 times longer in

some cases. Notably, the majority of tests (70%) exhibited shorter average execution times during failing attempts. This observation suggests a potential avenue for employing machine learning methods, similar to the approach of Lampel et al. [60], to classify failures as either flaky or non-flaky based on the test its own test execution time variations. However, we did not delve into this specific investigation in the current study. Furthermore, a comprehensive understanding of the distribution of normal test failures is necessary to effectively apply such machine learning techniques, which we leave as an area for future research.

Testing Device: We analyzed the specific virtual machines (VMs) and physical machines used to execute the tests to investigate the potential impact of data isolation and hardware-related issues. Exact may inadvertently introduce data isolation effects by rerunning tests on different machines. This phenomenon is analogous to the findings of Bell et al., who observed that restarting the test machine between test runs can impact test pass rates [11]. However, determining the impact of this data-isolation effect proved inconclusive in our analysis. This was primarily due to data clutter and the limited number of tests that failed more than once. Despite these challenges, we believe that information regarding the specific machines used for test execution could still hold significant value. We observed several instances where Exact engineers utilized this information to argue against test brittleness or test dirtiness.

Furthermore, our analysis of *all* test attempts and VM information did not reveal significant variations in flakiness across Exact's dedicated machines. However, we observed notable differences in flakiness rates when tests were executed on different platforms, specifically on AWS hardware compared to Exact's dedicated hardware (Section 4.1.3). This finding aligns with existing literature that emphasizes the resource-dependent nature of flaky tests [94]. While resource availability may differ between Exact's servers and AWS servers due to hardware variations, we expect resource availability to be relatively consistent within Exact's dedicated hardware environment, given their identical physical hardware (Section 2.2.2).

However, our analysis did not reveal a significant impact of hardware congestion on flakiness. We executed the maximum allowed number of concurrent pipelines and also conducted tests with no other pipelines running concurrently. These observations suggest either limited hardware congestion within Exact's VM setup or a minimal impact of such congestion on test flakiness. Since the only useful information source was the test-platform we do not classify and physical test runner information as useful and instead suggest that the less convoluted measurement of the test platform to be the more useful alternative.

Exception Type: The categorization based on error messages was more precise, rendering the categorization based on exception types redundant. We differentiated based on both error message and exception type if an exception was thrown.

Failure Moment: Investigating whether tests failed within the test method or within the generic test initialization showed that roughly 20% of tests failed within the generic initialization method. However, this categorization did not provide any benefits compared to error message categorization.

Representative Metrics for Flakiness at Exact

From the data obtained in our benchmarks, we identified the following metrics, overviews, and information types that Exact found valuable or that were utilized in our analysis.

FPPP (Flakiness Pipeline Pass Percentage): The FPPP proved valuable by quickly conveying the state of the system to technical management and accurately differentiating within which testing stage the impact of test flakiness was most prevalent. Due to its broad applicability for both developers and management, this metric is discussed in more detail in Section 5.1.5.

Test Attempt Information: The ability to see all normal failed test attempt information remained essential for the developers at Exact. In our experience, we observed that aggregated flaky test reports were used with various granular levels of information, eventually resulting in looking at individual failed test attempts. In meetings, where flaky-test reports were used as a high-level descriptor of the flakiness issue and its categories, Engineers still preferred to glance at individual failed test attempts grouped by specific information. This helped them verify the categorization or get an idea of the test names. We observed this in team meetings where they determine the next quarterly plans and used the flaky-test reports to estimate whether or not to focus on fixing flaky tests. We therefore think **it is essential when providing engineers with aggregated test information that they are able to switch between views and can see the individual test attempt information grouped by certain factors**, even if it is only to increase trust.

Test owners: Exact has many different teams all managing code within the main repository. Exact divides its test suite based on assemblies to establish which team should investigate the issue in case of a test failure within the RRT. We found that developers did not feel responsible if the failing tests were not within an assembly managed by their team. While this preconception regarding responsibility might not be valid as we have seen inter-assembly test brittleness, we do think it is a separation that is necessary in aggregated views to satisfy developers.

We further found that 76% of our observed flaky tests stemmed from 3 of the 42 teams responsible for 26% of the API and Integration tests. These 3 were part of the 7 teams with the highest number of tests they were responsible for; 71% of API and Integration tests and 90% of the respective flaky tests. Exact the company attributed this high density of flakiness within these teams to the fact that certain teams had more legacy code and needed to test more complex systems. This is in line with the findings of Gruber et al. that indicate that flakiness might be more prevalent within certain topics and more mature projects [36]. However, this is based on Exact the company its notion, and we leave validation open for future work. Furthermore, this situation illustrates the possibility of the increased number of flaky tests in larger development teams as stated by N. Mellifera [68].

Pass Rate: Categorizing flakiness based on its failure frequency offers several key advantages. Firstly, it empowers developers to more effectively assess whether flakiness has been resolved by comparing changes in failure frequencies or by observing the behavior of tests that consistently fail on the first attempt. Secondly, this approach facilitates a more targeted focus on the most impactful flaky tests when integrated with a test-attempt rerun strategy, since the less frequent flaky tests are better mitigated by rerunning.

We initially visualized test flakiness by tracking the number of attempt failures for the first, second, or third attempts, and ordering tests accordingly. This approach, combined with error-type categorization, was highly valued by Exact developers for its effectiveness in categorizing flaky tests. Building upon this initial approach, we subsequently introduced the concept of pass rates to further differentiate between levels of flakiness. This refinement proved valuable by

enabling us to filter out sporadic flakiness potentially caused by external factors and to focus our attention on tests that more frequently disrupt pipelines, as discussed in Section 4.1.

Our approach resonates with the practices of numerous large tech companies, including Apple [53], Ericsson [86], Facebook [65], Google [72], and Spotify [78] all of which have adopted various forms of flakiness scores to measure and visualize test flakiness within their respective systems. The rationale behind this widespread adoption is aptly captured by Facebook's observation: "All real-world test are flaky to some extend" [65]. This notion aligns closely with our own discussions in Section 5.2.3 and 5.2.4, underscoring the inherent presence of some degree of flakiness in any real-world testing environment.

Error Type: We were able to pinpoint most of the flakiness root causes based on the frequency of certain error types. These error types are at the center of this research, and we believe that most of our contribution to Exact's its flakiness improvements can be attributed to these categorized error types. They have been used for RQ0 to determine root causes and signal systematic issues leading to flakiness, as illustrated with 'Log' (Section 4.5.1) and 'User Not Active' (Section 4.2.1) errors.

However, since we defined the categories manually, this categorization of flaky tests based on error messages is an area that could be further investigated for more rigor and greater autonomy. We think that it would benefit a lot from incorporating work regarding bug report categorization who perform similar operation to facilitate developers in tackling bugs [12, 13, 122]. We discuss these possibilities in Section 6.2.

Stack Trace: Certain errors contained very cryptic error messages. The stack trace helped to categorize these. The 'SpecFlow Assembly Init' category (Section 4.1.3) only contained the error message 'Assert.Fail failed. One or more errors occurred.'. Based on the stack trace we were able to differentiate this error message and determine its root cause. The stack trace was often examined by the developers when viewing individual test attempts to understand the context of the failure and locate where the test failed.

Takeaway

Overall, allowing engineers at Exact to group flaky test information and investigate it with various levels of granularity, including individual failure information, significantly helps developers. We therefore suggest putting more focus on this ability to sift through the data with grouping, filtering, and other metrics to support developers in finding systematic issues and interdependencies between tests.

Not All Information is Equally Useful for Addressing Flakiness

We observed that categorized error types, individual test pass rates, and their associated failed attempt information, along with the Failure Probability per Pipeline (FPPP), proved particularly valuable for addressing test flakiness.
Implication: Incorporating these information sources into flaky test reports, tools, or detection methods can significantly enhance the understanding of test flakiness.

5.1.4 Information Motivates Developers to Fix Flaky Tests

This section explores the motivational benefits of flaky test information for developers. Similar to Coverity, Inc.'s approach of highlighting the 'top 10 new defect owners' to the development team to promote bug reduction [16], this study observed that providing clear, grouped, and quantified information about flaky test issues led to both reactive and proactive fixes by developers.

Interactions with Exact engineers revealed that information regarding flaky tests motivated both developers and management to address flakiness. Section 5.1.5 discusses how motivation through management, catalyzed by the ability to communicate the state of flakiness with the FPPP metric, influenced engineers to target flakiness. This section focuses on intrinsic motivation for developers.

Following a meeting with Exact developers to discuss overlapping categories and potential root causes of flakiness based on our initial benchmarks, some developers began investigating and addressing categories where they had hypotheses about the root cause. Notably, they proactively pushed out patches for these flaky tests, even before planned task division or issue assignment. Informal inquiries revealed that the developers found fixing systematic flakiness issues, which can impact multiple tests, more intrinsically rewarding than fixing individual flaky tests.

This example highlights the motivational benefits of providing developers with clear and actionable information about flaky tests. Based on similar interactions with Exact developers, we believe that information-induced motivation significantly contributed to the overall improvements in flakiness observed throughout this study (Section 4.13).

Information Motivates Developers to Fix Flaky Tests

Information regarding flaky tests motivated developers to target and fix root causes of flakiness and helped illustrate the necessity of addressing flakiness to management.
Implication: Providing aggregated information about flakiness can indirectly significantly reduce flakiness in your systems. It does so by not only helping developers but also by motivating both developers and management to target flaky tests.

5.1.5 The Impact of FPPP

While other metrics can convey the state of the flaky system, the Flakiness Pipeline Pass Percentage (FPPP) directly quantifies the impact of flakiness on possible key business values. Prior to the introduction of the FPPP, it was unclear whether flakiness was the primary cause of the low *successful release rate* at Exact. Other factors, such as careless code merges or a lack of accountability for RRT failures, could have contributed to the problem. The combined FPPP for API and Integration tests was a mere 25%, or 27% excluding UI tests, indicating that only one in four pipeline runs passed successfully, even when the system was deemed correct. This correlated with the then-current 60% release rate, considering that Exact executed roughly 4-5 RRTs per day.

Consistent with prior research, and in correspondence with our own findings (Section 4.5), no direct correlation exists between the number of flaky tests and the number of failed builds

[55]. The FPPP effectively overcomes this limitation by directly quantifying the likelihood of a build failing due to flakiness. Which in turn directly describes the effect of Exact its business values, the *successful release rate*. Since the start of this study we have observed the FPPP and the *successful release rate* to behave in accordance with each other as illustrated in Figure 4.13 and Section 4.5.3. This indicates that the FPPP not only quantifies the impact of flakiness on build stability but also serves as a crucial predictor of the successful release rate, a critical business metric at Exact.

Furthermore, the FPPP metric effectively communicated its impact to both developers and management by making its implications tangible. While managers were aware of flakiness issues, they were surprised by the low FPPP. Sharing this number effectively conveyed the problem, leading to discussions on strategies and division of tasks to prevent test flakiness in upcoming sprints.

The Impact of FPPP

The Flakiness Pipeline Pass Percentage (FPPP) metric tangibly quantifies the impact of flakiness for both developers and management.
Implication: Regularly monitoring the FPPP can provide valuable insights into the impact of flakiness and help identify the need for proactive measures to address this issue.

5.2 The Many Faces of Flakiness in an Industrial Setting

'Test flakiness' is a collective term for the observable behavior of non-deterministically failing or passing tests, caused by multiple different bugs, mistakes, or issues. Consequently, there is no single solution to fix it, and it behaves differently in per environment. In our case study at Exact, we observed several distinct ways in which flakiness manifests within a database-heavy industrial system. The results include relevant examples based on the effects of the proposed mitigation strategies. This section builds upon these examples and discusses various ways how flakiness can exhibit and its implications.

Section 5.2.1 demonstrates that flakiness exhibits distinct characteristics across different types of tests or test suites. For instance, the frequency and nature of flaky tests may vary considerably among API, Integration, and UI tests at Exact. Section 5.2.2 presents the intricate relationships between different types of tests related to flakiness, such as flaky, dirty, offending, and brittle tests. It emphasizes that these categories are not mutually exclusive and that issues within one category can indirectly impact the behavior of other tests, leading to unexpected failures or consistent failures of seemingly unrelated 'correct' tests. Section 5.2.3 discusses how even the test environment itself can be inherently flaky, leading to unpredictable test failures that can potentially affect every test. It also explains why reruns can be a valuable mitigation strategy in such scenarios. Finally, Section 5.2.4 addresses the evolving nature of flakiness. It highlights the risk of an ever-growing set of labeled flaky tests, as all tests can potentially exhibit non-deterministic behavior due to factors such as dirty tests or environmental instability. This over-labeling can dilute the effectiveness of the 'flaky' classification and hinder effective analysis. Therefore, we emphasize the crucial need to differentiate between truly impactful flakiness and non-impactful occurrences.

5.2.1 Flakiness Exhibits Differently across Test Suites and Environments

Flaky tests exhibit diverse behaviors, stemming from various underlying causes. For example, some tests demonstrate varying degrees of susceptibility to resource changes, ranging from none to significant [94]. Additionally, flakiness can be categorized as order-dependent or non-order-dependent [9, 74, 119], and language-specific quirks, such as the random ordering of certain collections in Python 3.2+ [36], can also contribute to flakiness.

Importantly, these flakiness categories are not mutually exclusive. In line with this observation, Eck et al. found that when developers were asked to categorize 200 flaky tests, multiple tests were often assigned to more than one category [23]. This section illustrates the intricate interplay of various factors affecting flakiness and highlights how any change to the suite and environment can cause flakiness to manifest differently in various test suites.

While Eck et al. describe the flaky category 'platform dependency' as failures related to specific operating systems, the observations in this study align more closely with the concept of 'environment dependency', which encompasses factors such as hardware, network conditions, and cloud infrastructure. This finding aligns with the concept of Resource-Affected Flaky Tests (RAFTs) introduced by Silva et al. [94], where test outcomes are influenced by resource contention, availability, or limitations.

Interestingly, we observed RAFTs manifesting in two distinct ways, likely due to these tests also being order-dependent and, in some instances, even dirty due to language-specific non-deterministic quirks in the VB and C# .NET Dispose pattern, such as race conditions during garbage collection. The two observed manifestations of flakiness are as follow:

1. **Lowered Pass Rates for Specific Tests**: Certain tests exhibit consistently lower pass rates on specific platforms, particularly observed within Exact's Integration tests.
2. **Constant Overall Pass Rate Impact with Ever-Changing Responsible Tests**: A group of tests may experience consistently lower pass rates on a particular platform, but the specific tests within that group may vary. This behavior was more common in API test compared to Integration tests at Exact.

The test environment itself significantly impacts test flakiness. The study experiments were conducted within a large-scale, production-like environment, as detailed in Section 3.1.1. While we aimed to minimize external factors, maintaining a realistic test environment necessitated some compromises (Section 5.3). Interestingly, we observed patterns in flakiness differences when running tests on AWS cloud services compared to dedicated hardware.

The Test Platform Impacts the Flakiness Frequency for a Specific Group of Tests

We observed that a specific group of tests linked to 'SpecFlow Assembly Init' failed more often on AWS hardware compared to Exact's dedicated hardware (Section 4.1.3). Interestingly, this behavior was consistently more prevalent in benchmark runs where at least the dirty row-count tests were disabled. This shows that this increased flakiness exhibited within a specific group of tests is dependent on both the test platform and some interdependency between the disabled tests and the unaffected flaky tests.

Further investigation into the flaky nature of these tests revealed that they subsequently failed with other errors and causes. In some instances, these tests resulted in three failed attempts in a

pipeline, thus resulting in a pipeline failure. This illustrates how a combination of flaky factors can cause pipeline failures and that one **test can exhibit different types of flakiness influenced by various types of factors, sometimes even several at the same time, impacting flakiness**.

This example also shows how 'correct' changes in the test environment, such as disabling dirty tests, can seem to have adverse consequences, such as increasing the failure probability of unrelated RAFTs. However, we found no data or brittleness relation between the disabled tests and the more likely 'SpecFlow Assembly Init' failing tests. The 'SpecFlow Assembly Init' errors indicate a local server being offline or not responding fast enough. We hypothesize that this is caused either by non-disabled tests that hold locks to the databases, which are now more likely to occur before the 'SpecFlow Assembly Init' tests due to the disabling of a significant percentage of tests, or some underlying server caching that used to be triggered by the disabled tests. This demonstrates how 'correct' or 'normal' interactions with unrelated tests can have adverse consequences on other 'correct' tests due to unforeseen interactions, resulting in an increase in observed flakiness.

Changes in the Test Environment Amplify Flakiness Unrelated to Particular Tests

API tests exhibited more sporadic flakiness on AWS hardware, particularly due to timeouts (Section 4.1.3). The tests responsible for the attempt failures kept varying, indicating that the root cause affected by the platform change might be related to Exact's test environment configuration or framework, or the system under test. This indicates that although Exact mitigated flakiness by minimizing resource contention with dedicated VM environments hosting all related services and sequential test execution, the test suite still experiences systematic test suite-related flakiness amplified by changes in the test platform.

Within the ('WeDispose') implicit dispose benchmark, we also observed tests potentially impacting other tests. One group of tests described in Section 4.3.2 exhibits a 'Fire and Forget' test smell [31], where threads are started but not properly joined, leading to potential resource leaks and subsequent test failures. These tests can cause flaky test-attempt failures due to timeouts or dirtiness in any tests that try to access the same database before the threads terminate. However, these same tests caused consistent test failures in the 'Post-Disposing' benchmark, implying that the same test is also a type of test that can fail due to a non-deterministic language-specific quirk from the VB and C# .NET Dispose pattern. For implicit dispose, its flakiness manifestation depends on execution speed, whether other tests use databases, and thereby test execution order or database locking configuration. This indicates how a combination of flaky natures ('Async Await', 'Platform Dependency' [23]) and influencing factors (language, test platform, and execution order) prevalent within one test (or group of tests) can change how often and where flakiness exhibits.

Flakiness Exhibits Differently across Test Suites and Environments

Flakiness has a multifaceted nature and it exhibits diverse characteristics across different types of tests, platforms, and environments. Consequently, changes in the test suite or test environment can impact flakiness in both correct and incorrect tests.
Implication: There is no '*one solution fits all*' for flakiness and Engineers need to understand the intricate relationships that contribute to flakiness.

5.2.2 Differentiating Flaky, Offending, and Dirty Tests

There were instances of dirty tests at Exact, which can consequently lead to flaky or brittle test behavior. Contrary to common assumptions of Exact developers, flaky tests are not always the primary culprits of test failures. Exact relies heavily on tests to perform their own cleanup rather than their own setup. This approach accelerates test execution and simplifies the setup phase. However, it can increase the difficulty of debugging and introduce order dependencies between tests. This section explores the relationships among flaky tests, dirty tests, and what we term '*offending tests*'. It analyzes the benefits and drawbacks of relying on tests for self-cleanup and emphasize the importance of clearly distinguishing when a test exhibits 'offending' behavior.

We define '*offending tests*' as those that violate the system's (in this case, Exact's) expectations of correctness. In Exact, all tests that leave global settings in a dirty state are considered offending, as this can lead to various unforeseen issues. However, tests that leave log-related database tables dirty are permissible and even encouraged in some cases for debug reasons (Section 3.2.3). While these non-offending tests may still contribute to flakiness, the responsibility for handling the dirt in the log tables lies with the test that is flaky because of it. In contrast, for other types of dirtiness resulting in a flaky test, such as a through dirty global settings, the dirty test is then the offending entity.

Methods to Achieve Test-Independence are Unpractical

Similar to other systems, Exact exhibits order dependencies between tests resulting in flakiness. Shi et al. identify two types of tests affected by order dependencies: victims and brittle tests. Victim tests pass when executed in isolation but fail when run after a specific dirty test. Brittle tests, conversely, fail in isolation but pass when executed after a specific dirty test [92]. (1) Running tests in isolation or (2) with a fixed test-order is impractical within Exact its industrial system due to the significant overhead involved.

The first method, running tests in isolation, is infeasible due to the considerable overhead, as illustrated in other research [9, 74]. Alternative approaches that attempt to leverage or fix test-order are also unsuitable for Exact, as Exact prioritizes minimizing the longest test-runner execution time in a pipeline. Consequently, Exact employs a multi-test-runner approach with a dynamic test scheduling system, as described in Section 2.2.

The second method, a fixed test-order, might incur overhead due to suboptimal test execution sequences. The scheduling problem, which involves determining the optimal order for executing tests, is a well-known NP-complete problem [105]. To address this, Exact employs an alphabetical dynamic first-in-first-out (FIFO) scheduling system. This approach, while constraining the total execution time to at most the difference between the longest and shortest job, exceeding the optimal time [33], introduces variability in the test execution order.

These arguments combined demonstrate that true test independence is unattainable in the industrial setting of Exact, and therefore are order dependencies between tests inevitable. Based on feedback from Exact engineers, replicating flaky tests is challenging, and **the ability to replicate or at least observe the test-order within a pipeline is crucial**.

Dirty Tests Cause Nasty Bugs

Dirty tests are not inherently incorrect and rarely lead to flaky tests. In line with Shi et al.'s findings, we observed both brittle and victim tests within Exact [92]. However, contrary to Shi et al.'s findings, the brittle test are more prevalent than victim tests at Exact (Section 4.3.2). Interestingly, despite the presence of numerous dirty tests (over 10% after filtering allowed dirty tests; Section 4.4.1), fewer than 1% of all Api and Integration tests were flaky, and an even smaller percentage were brittle or victim tests due to dirty tests, according to the benchmark results when disabling both groups of dirty tests (Section 4.4). This observation highlights that while dirty tests are common, not all are offending (as some dirtiness was allowed) or result in flaky test behavior.

However, offending dirty tests *can* be highly problematic and difficult to detect. The benchmarks related to RQ2 provide an illustrative example: Several offending singleton-dirty tests caused 96 constant first test-attempt failures (Section 4.3.2). These results consistently demonstrate the complex interdependencies between certain tests and the potential for dirtiness to propagate across assemblies and impact seemingly unrelated tests.

Offending Test Guidelines are Needed

We believe that a clear definition of when a dirty test is considered violating is essential. Both approaches, relying on tests for self-cleanup (dirty test is offending) and relying on tests for self-setup (flaky test is offending), have their own advantages and disadvantages; When regarding the flaky test as offending, the bug is localized to that specific test. Conversely, identifying the dirty test as offending enables greater abstraction of the test setup, as observed within Exact. Exact lacks a strict set of rules regarding permissible database dirtiness, which likely contributes to the high number of tests that leave the database in a dirty state.

Although dirty tests generally do not result in flaky tests, as our findings indicate, one dirty test can lead to hard-to-find bugs or flakiness in a larger number of tests. Establishing **clear guidelines for the data that a test must clean enables development of tools, such as DB sanity checks, that handle dirty tests appropriately and prevent flakiness during development**. This differentiation also plays a crucial role in determining which tests should be disabled and whether a test failure signals a genuine system issue. At Exact, engineers typically evaluate the flakiness of a test responsible for an RRT failure by executing it locally in isolation. If the test passes in isolation, it is often deemed flaky and disabled before the subsequent RRT run. This approach, however, presents significant drawbacks, as exemplified by the following two subsequent scenarios:

- **The test passes in isolation**: In this scenario, the flaky (non-offending) test, which correctly verifies system behavior, is mistakenly removed. Conversely, the dirty (offending) test, the true source of the flakiness, remains within the test suite. This can potentially trigger subsequent test failures. This practice mirrors the early methodology employed by Facebook [7] of deleting test as long as the production system works. However, this has been identified as a detrimental approach leading to tests diminishes the test suite's coverage and jeopardizing the reliability of the system [85].

- **The RRT-failing test is brittle and therefore fails in isolation**: When the RRT-failing test also fails in isolation, developers often erroneously attribute the failure to a genuine system issue. Consequently, development efforts are misguided towards investigating the root cause of the isolated test failure. However, in this scenario the test's brittleness, rather than a system fault, is the primary reason for its local failure and the RRT failure might still be attributable to flakiness.
  This scenario presents several drawbacks. Firstly, developers tend to utilize the local isolated failure for debugging, leading them to investigate the brittle test failure, which may stem from a different cause than the RRT failure. Secondly, developers might also analyze the commit history to identify the source of the issue. Since the test's brittleness might have predated the last successful RRT execution, this analysis can lead to investigating irrelevant code changes. These misdirected efforts can ultimately result in wasting the limited time available before the next RRT execution, potentially jeopardizing release schedules.

Both scenarios can be addressed by focusing on the offending test rather than the flaky tests themselves. In the first scenario, guidelines should prioritize the offending test, recognizing that the flaky test may be valid and should remain in use. In the second scenario, the knowledge of a potentially brittle test can be leveraged within the debugging methodology. This involves not only running the failing test in isolation on the failing RRT commit but also on the last working RRT commit. This approach helps determine whether the test was already brittle before any code changes, enabling a quicker assessment of whether the RRT failure signifies a genuine system fault.

Consequently, clear understanding whether a flaky, victim, or brittle test is offending is crucial for preventing the unnecessary filtering of 'correct' tests and avoiding the waste of development time investigating test failures unrelated to system faults.

Differentiating Flaky, Offending, and Dirty Tests

A flaky test does not necessarily indicate an incorrect test. Dirty tests can cause numerous 'correct' tests to exhibit difficult-to-debug flaky behavior. Project teams should determine which dirtiness is acceptable within their system and which results in an offending tests. This facilitates appropriate and strict handling of the dirtiness accordingly and ultimately improves development productivity.
Implication: Establishing clear distinctions between offending and non-offending tests and adapting the test framework and workflow accordingly accelerates system verification and minimizes the effort spent on addressing non-offending tests.

5.2.3 The Test Environment is Flaky

We observed instances of widespread test failures that appeared to be independent of the specific test being executed (Section 4.1.2). Specifically, we observed 792 tests failing with a 'Canceled Task' message across two API test runs. Further investigation revealed that this error extended beyond the VM, as we observed a concurrent increase in randomly failing tests within the Integration test stage. While the precise cause of this error remains unknown, it likely originated

from the physical machine. Although Exact utilizes consistent hardware to reduce hardware-related flakiness, these instances still occurred. This observation supports the assertion by Zhang et al. [119]; Uncontrolled factors can *always* introduce flakiness into a system.

These are not isolated incidents. There were similar outliers in the post-'DB background task minimization' benchmark run (Section 4.2.2), where 110 tests from the same assembly failed in only 2 out of 32 pipeline runs, all exhibiting related error messages most likely caused by a timeout. These instances demonstrate that inherent external factors can contribute to test flakiness. While the true nature of these errors remains uncertain, and they might stem from simple underlying configuration issues, they highlight a crucial point: These test failures are irrelevant for developers who attempt to regression-test their code changes. Implementing strategies such as test reruns can potentially save resources and development time by mitigating the impact of these unpredictable occurrences.

The Effects of the Rerun Strategy

Sentiment within Exact regarding test rerunning was divided. Some argued that rerunning tests allows flaky tests to persist within the system, masking underlying issues rather than addressing them – a perspective aligned with Vassallo et al., who classified test retries as a 'CD smell' [107]. Supporting this view, we did not observe any flakiness within the 42,000+ Unit tests, which adhere to a zero-flakiness policy (i.e., no test reruns), in any of our benchmark runs. However, this observation might be attributed to potential proactive deletion of flaky tests whenever encountered by developers, an approach similar to earlier practices at Facebook [7] which is not beneficial for the long term health of the system (Section 5.2.2).

Moreover, we found that a more nuanced approach is necessary for more integrated test stages, such as API, Integration, and UI tests. These tests exhibit a higher degree of complexity, relying on interactions with databases and services. As demonstrated by that almost all test where catagorized as flaky by Azure DevOps (Section 3.1.1), these tests are more accurately characterized by the "Assume All Tests Are Flaky (ATAF)" principle [39] which we will illustrate in more detail in the next section. Therefore, do we believe that test retries can offer valuable benefits in industrial settings where factors beyond the control of individual developers, such as environmental fluctuations or transient hardware issues (Section 4.1.2, 4.1.3, and 4.2.2), can significantly impact test stability. Reruns can effectively mitigate the impact of these unpredictable factors, enhancing the reliability of the CI/CD pipeline and reducing the time wasted on false failures. This not only improves developer productivity but also minimizes the frustration associated with intermittent test failures. Furthermore, reruns can serve as an early warning system, signaling potential underlying issues within the test environment or infrastructure, as evidenced by the transition to AWS (Section 4.1.3). By analyzing patterns in flaky tests, particularly those that consistently fail on the first attempt but pass on subsequent retries, we can gain valuable insights into their issues, enabling proactive measures to improve the overall stability of the testing environment.

Therefore, while test **retries** should *not* be considered a substitute for addressing the root causes of flakiness, they **can serve as a valuable mitigation strategy in industrial settings** where complete elimination of all flakiness may not be feasible. With as essential detail that they don't substitute flakiness addressing policies and strategies. Unrestricted use of retries

can have adverse consequences, such as deeming flakiness issues of lower importance when they are less visible. This phenomenon is evident in our observation of a rapid increase in the number of failed test attempts with a high FPPP (Section 4.5.2). Consequently, the successful implementation of a test rerun strategy necessitates its integration with robust monitoring and measurement mechanisms. These mechanisms, similar to those employed by Coverity [16] or Mozilla [60], can effectively track issues and prompt developers to address the root causes, ensuring that test retries are used strategically and do not hinder the overall goal of improving test stability and reliability.

The Test Environment is Flaky

The test environment itself can exhibit inherent flakiness. Instances such as widespread test failures due to service outages, failed container starts, or host machine crashes can impact the entire test suite.
Implication: CI/CD engineers and other developers should be aware that certain errors may propagate throughout the entire system. In such instances, where the flakiness appears sporadic rather than systematic, it may be more effective to simply disregard these results and attempt a rerun.

5.2.4 The Ever-Growing Set of Flaky Tests

Our findings reveal a concerning phenomenon: the ever-growing set of flaky tests. This phenomenon arises from the interdependent nature of tests within a system, where issues in one area can propagate and impact the behavior of other tests. As illustrated in the previous two sections, a flaky test does not necessarily indicate an incorrect test, and flakiness can manifest in any test due to factors such as dirty tests or environmental influences.

This ever-growing set of flaky tests was particularly evident within Exact, where almost every test was initially labeled as flaky by Azure DevOps's default flaky flagger (Section 3.1.1). This widespread labeling led engineers to overattribute test failures to test flakiness. Our findings are corroborated by other research efforts that have adopted the Assume All Tests Are Flaky (ATAF) principle [39, 65] or have demonstrated that flaky tests can remain undetected even after 10,000 runs [4, 11, 56]. This over-labeling of tests as flaky **dilutes the impact of the 'flaky test' classification and hinders effective analysis**. Therefore, we believe it is crucial to differentiate between truly impactful flakiness and non-impactful occurrences.

Differentiating Flakiness Methods

This work differentiated between flaky tests based on their *pass rates*, distinguishing between 'sporadic' and 'non-sporadic' flakiness (Section 4.1.1). This approach proved beneficial, aligning with developers' perceptions of the most 'problematic' flaky tests – those that frequently result in test and pipeline failures. The analysis revealed that only 25% of flaky tests were deemed non-sporadically flaky, yet these tests were responsible for 72% of all test failures,[1]

[1] A test failure in the context of Exact means 3 out of 3 failed attempts within one pipeline.

demonstrating the significant impact of these highly flaky tests. While the pass rate differentiation approach may not be universally applicable, we emphasize the importance of establishing meaningful criteria for classifying and prioritizing flaky tests. Other potential methods include the following:

- Manual classification frameworks: Providing a framework that allows developers to categorize flaky tests based on their observed flaky behavior and potential causes.
- Classification based on recent flakiness: By considering flakiness only based on the last x number of test reruns, this approach allows for categorization across different commits and gradually phases out fixed or non-recurring flaky tests. Most similar to the approach of Kowalczyk et al. [53], with many others also utilizing the history of flakiness for flakiness differentiation [65, 72, 78].

However, it is crucial to avoid over-reliance on code change analysis alone. For instance, a method that leverages the version-control system or coverage analysis, similar to approaches used in other work to distinguish between flaky test failures and actual system errors [11], would not effectively detect flaky behavior caused by environmental changes or modifications to external dependencies. Consequently, this approach may still result in an ever-growing set of tests labeled as flaky.

Less Impactful Flakiness Is Still Problematic

Although we emphasize the need to differentiate between levels of flakiness based on impact, this does not imply that less impactful flakiness should be entirely disregarded. It is important to recognize that even tests with relatively high pass rates (e.g., above 0.9) can still have a significant impact. These tests, while less frequent, can contribute to wasted resources due to unnecessary rerun attempts and pipeline failures.

Moreover, when combined with other factors, such as brittle tests or external dependencies, the impact of these seemingly 'less problematic' flaky tests can be amplified. For example, if a brittle test (which relies on the execution of another test) fails on its first attempt due to external factors, it is unlikely to pass on subsequent attempts because the other test (on which it relies) may not be executed beforehand during the rerun.

Another example involves the 'SpecFlow Assembly Init' tests that failed on subsequent attempts with different errors (Section 5.2.1). The individual test failure probability of each instance of flakiness is low due to the likelihood of passing on a rerun. However, when combining the probabilities of these seemingly independent events, the overall impact on test stability can be significant.

Takeaway

In conclusion, considering every instance of non-deterministic behavior as a distinct 'flaky' test can lead to an overabundance of such labels. This diluted the impact of the 'flaky' classification within Exact, leading engineers to misattribute non-flaky test failures to flakiness. Therefore, we believe it is essential for future research and industry practices to differentiate between levels of flakiness based on their frequency, relevance or impact.

The Ever-Growing Set of Flaky Tests

All tests can eventually be deemed flaky due to the influence of dirty tests or the inherent flakiness of the test environment. This can lead to an over-reliance on the 'flaky' label, diminishing its effectiveness in identifying truly impactful issues.
Implication: Quantifying the flakiness of tests using metrics such as pass rate or distinguishing between sporadic and non-sporadic flakiness helps to illustrate the true extent of flakiness.

5.3 Threats To Validity

This study was conducted within a single, live, industrial setting, investigating flakiness, which inherently involves uncertainty in its manifestation. The decision to control certain factors necessitated a careful balance between internal and external validity. For example, to enhance the ecological validity of the research, we executed all tests precisely as they occur within Exact's CI pipeline, including variations in test order. While this approach increases the relevance of our findings to real-world scenarios, it may introduce external factors that could potentially compromise internal validity. Although this design choice also enhances content validity by ensuring that we accurately measure all forms of flakiness within Exact, it may potentially compromise the construct validity of the experiments that investigate the benefits of automatic approaches by allowing external factors to influence the results.

This section examines potential threats to the validity of our study, drawing upon the framework outlined by Wohlin et al. [116]. We critically analyze potential biases and limitations that could have influenced our results, such as the potential for missed flaky tests, the limitations of our analysis tools, and the specific context of our study within Exact's software development environment. By addressing these potential threats upfront, we aim to provide the reader with a more informed and critical perspective on our research findings.

5.3.1 Missed Flaky Tests

A fundamental challenge in this research is the potential for missed flaky tests. We mitigated this by rerunning each test multiple times within each benchmark run, minimizing the effects of infrequent flaky tests and other confounding factors such as test dirtiness or resource congestion, as observed in other work when only rerunning the flaky tests [55]. Nevertheless, the possibility of undetected flaky tests remains.

Prior research has demonstrated the difficulty of identifying all flaky tests, even with extensive reruns (e.g., 10,000) [4, 56, 11]. This limitation directly impacts the internal validity of our findings, potentially leading to an underestimation of the true extent and impact of flakiness within Exact's CI/CD pipelines. Consequently, we may obtain inaccurate assessments of flakiness and flawed conclusions about root causes, hindering accurate evaluations of the effectiveness of our mitigation strategies.

To mitigate this, we investigated the amount of new and change in information after at most 36 runs. We observed that the majority of flaky tests and non-sporadic flaky tests were identified

within the first 5 pipeline runs. However, we conducted 26 pipeline runs in most benchmarks to ensure stability and negate the effects of other confounding factors such as external influences or platform changes.

Bias Toward Sporadic Flaky Tests and Limitations of Manual Inspection

By treating all 'flaky' tests equally, we obtained a skewed and potentially impact-diluted view of flakiness (Section 5.2.4). This threatens the construct validity of the research by misrepresenting the impact of the changes and the influence of external factors on flakiness. To address this, we focused on the average number of flaky tests per pipeline and filtered based on pass rates (distinguishing between sporadic and non-sporadic flaky tests), as described in Section 4.1.1.

However, fully filtering based on pass rates can introduce a bias, potentially threatening the internal validity of the findings. By prioritizing the identification and analysis of non-sporadic flaky tests, we may underrepresent less frequent or more subtle types of flaky behavior. For instance, order-dependent flaky tests exhibit flakiness only when executed in a specific order. The varying test-order across pipeline runs can mask their flakiness, leading to potentially higher observed pass rates and underestimating their prevalence. This bias threatens the internal validity of the research by potentially leading to an incomplete picture of the root causes of flakiness within Exact's CI/CD pipelines. To mitigate this, we always jointly investigated the overall average effects on flakiness alongside the individual movements of sporadic-flaky tests.

An alternative approach to identifying sporadic flakiness would be to employ statistical tests for each individual test. However, as outlined in the results and discussion, hidden and unknown dependency relations exist between tests. Therefore, many statistical tests that rely on observation independence or monotonic relations are not applicable without threatening the construct validity of the research.

Conclusion

By missing these less frequent types of flakiness, we may have an incomplete understanding of their root causes and their impact within the system, threatening the internal validity of the research. Moreover, our findings may not be generalizable to other systems where these less frequent types of flaky behavior are more prevalent, threatening the external validity of the research.

5.3.2 Limitations of Sanity Checks and WeDispose

The accuracy of the analyses for RQ2 and RQ3 is inherently tied to the effectiveness of the WeDispose and sanity check tools. These tools, however, exhibit certain limitations that pose threats to the validity of our findings.

Both WeDispose and the current sanity checks may not identify all instances of violating code. For example, WeDispose relies on branchless code analysis within tests, potentially missing implicit dispose scenarios in production code or tests with conditional execution paths. Similarly, the sanity checks, despite their efforts, cannot detect all forms of test dirtiness. Furthermore, neither tool is currently adapted for SpecFlow-generated tests.

Some of these limitations stem from deliberate trade-offs to ensure the feasibility of our analysis in a real-world setting. This necessitated prioritizing enhanced applicability, potentially leading to an underestimation of the overall impact of test dirtiness. The following sections discuss these missed instances and illustrate their effects on the study's validity, along with the relevant mitigation or inspection methods employed to estimate or limit this impact.

Missed Dirtiness by Sanity Checks

Manual inspection revealed instances where objects and values are set at the test class or assembly level but not properly cleaned up after test execution. These persistent values can pollute the database and go undetected by the current sanity checks, which only examine individual tests. This subsection illustrates three types of cases missed by the sanity checks, resulting in overlooked dirty tests, thereby undermining the internal validity of our research.

1. **Dirtiness Occurs Outside Test Scope**: The current sanity checks primarily focus on individual test methods. They may miss instances where dirtiness is introduced by the following:
    - **Assembly or Test Class Initialization**: Our sanity checks are limited to the test method level, and dirtiness introduced at the class or assembly initialization level goes undetected. To illustrate the potential impact, if every, and only, implicit dispose were to leave the system dirty, then 3% of instances would be missed when only investigating dirtiness at the test level. This is illustrated by the fact that 3% of implicit disposals refer to missed disposals in the `ClassCleanup` (Section 4.3).
    - **Concurrent Test Execution**: Dirtiness from tests levering multi-threading can introduce unexpected data inconsistencies in other threads after the main thread terminates. We identified six such possible instances in our 'Post Disposing' benchmark (Section 4.3.2).
2. **Data Loss During Sanity Checks**: Both comparison methods employed in the DB sanity checks (MD5 hashing and total row count) can suffer from information loss. While the probability of MD5 collisions is generally low, it cannot be entirely ruled out.
3. **Abnormal Test Path**: Tests that abort abnormally (e.g., due to exceptions) or follow different execution paths on different attempts (e.g., due to dirty data) may not be properly evaluated by the sanity checks. This can lead to missed instances of test dirtiness.

To assess the impact of these limitations, we conducted an additional analysis with two more pipeline runs. While this analysis is also subject to an undervaluation bias, our findings suggest that the undervaluation of dirty tests is likely to be minimal. We identified 12 tests in group 3 that were not evaluated correctly due to abnormal termination. Additionally, 26 tests exhibited flaky DB sanity check behavior, potentially due to reasons from group 1 or 3. We considered these tests as dirty in the analyses of experiments and results of RQ3. When comparing unhashed or individual rows instead of total row and hash, we did not find any additional tests. These instances combined indicate that we may have potentially missed at least 38 tests, which is

approximately 0.2% of the total number of evaluated tests. This suggests that while our sanity check may not identify all instances of dirty tests, its overall impact on the findings is small.

Limitations of WeDispose

WeDispose currently focuses on identifying implicit dispose within branchless test methods (Section 3.2.2). This limitation may lead to an underestimation of the true prevalence of implicit dispose, as it may miss instances within tests with conditional execution paths. Furthermore, WeDispose primarily targets implicit dispose within test code. According to Microsoft, creating a Roslyn rule to detect implicit dispose in both test and production code is considered infeasible due to the high computational complexity [40, 52].

We have mitigated the impact of incorrect refactors on the internal validity by manually verifying all refactors. A second review revealed that only 1 out of 5,024 refactors was incorrect, and this instance is discussed in the results (Section 4.3). We further believe that the current WeDispose implementation represents an acceptable trade-off between effectiveness and feasibility, particularly in real-life scenarios. This effectively prioritizes ecological validity over internal validity.

SpecFlow Tests Are Not Evaluated by Our Automatic Tools

SpecFlow tests, which comprise approximately 20% of the codebase (Section 2.2.2), are not currently evaluated by the WeDispose or DB sanity check methods. This exclusion poses a potential threat to both the internal and external validity of our research.

- **Internal Validity**: The omission of SpecFlow tests may lead to an underestimation of the overall impact of test dirtiness within the system, as a significant portion of the test suite remains unanalyzed.
- **External Validity**: Our findings may not be directly generalizable to systems that heavily rely on SpecFlow or other specialized testing frameworks. Their unique structure, separating imperative and declarative aspects of testing, may inadvertently lead developers to overlook side effects and cleanup procedures, potentially increasing the prevalence of dirty tests. Section 5.3.3 discusses the applicability of these findings in more detail.

This limitation underscores the importance of considering the specific characteristics of different testing frameworks and their potential impact on test flakiness.

5.3.3 Exact's Framework Bias the Results

The results obtained from the experiments exhibit bias due to the specific context of Exact, potentially threatening the external validity of the research. All of the results and main takeaways presented in Chapter 4 and 5 are inherently tied to Exact. This implies that the results may not be directly applicable to other systems, potentially limiting the generalizability and thereby ecological validity of our work. To mitigate this, we employed an implication-based approach in all main takeaway boxes. Furthermore, we have extensively explained individual instances resulting in test failures, aiding in understanding the underlying behaviors of test flakiness and facilitating application in other projects.

Furthermore, several of the following factors specific to Exact can influence the findings in ways that may not be generalizable to other software systems:

- **Framework-Specific Setup**: Exact's framework and configuration impact the frequency of certain errors. This could lead to an overestimation or underestimation of flakiness levels compared to systems with different architectures, configurations, or development practices. This may be particularly significant in the findings regarding the impact of redundant background tasks (RQ1) and Exact's utilization of templated archtypes users to facilitate test data.

- **Exact-Specific Failure Categories**: The categories used to classify test failures are specific to Exact. This reliance on specific categories limits the comparability of our findings with other studies. To mitigate this impact, we only used categories to group errors and improvements or to explore data. Whenever we used these categories in our discussion of findings, we also mentioned the broader categories they are applicable to, enhancing their relevance to other contexts.

These factors pose a threat to the population validity of the research, as the findings may not accurately reflect the broader software development landscape. To mitigate this effect, we clearly outlined the specific details relevant for each research question in Chapter 3. We also outlined the specific Exact setup in Section 2.2 and provided a generalization of some specific testing issues encountered at Exact to facilitate the applicability of the results (see Section 2.2.5).
While this research, as a standalone study, might be susceptible to sampling bias due to its focus on a single framework, it contributes to a reduction in the overall sampling bias present in the literature. Existing literature predominantly concentrates on large technology companies or open-source projects (Section 2.1.1). Our work expands this sample set by exploring challenges within a typical *Software-as-a-Service* (SaaS) industrial software environment. To mitigate the potential for sampling bias, we integrated findings from other relevant studies into the explanations of the root causes of flakiness identified in RQ1, RQ2, and RQ3 (as determined by RQ0) in Section 3.

Effects of Different Software Languages

Different programming languages exhibit varying levels of flakiness due to inherent language characteristics, common coding patterns, and available testing frameworks. As demonstrated by Google's research [72] and the varying findings of research targeting different languages [23, 36, 55, 64], there are differences between flakiness in various languages. Our research at Exact primarily focuses on .NET with VB and C# for SQL Server and MSTests (Section 2.2.2). This reliance on a limited set of languages poses a threat to the external validity of the findings by limiting their generalizability to a broader software development landscape. However, given that most prior research has investigated flakiness in Java or Python (Section 2.1), our work, when considered in conjunction with other research, extends the generalizability of existing knowledge.

5.3.4 Experiments Conducted in a Live Environment

Conducting experiments in a live industrial environment presents a trade-off between external and internal validity. While it enhances external validity by providing insights into a real-world setting (ecological validity), it also introduces several challenges that can affect the internal validity.

Constant System Changes

In a dynamic business setting like Exact, the system is constantly evolving through ongoing development, deployments, and infrastructure changes. These changes can introduce unexpected variations in test behavior, making it difficult to isolate the effects of specific interventions or to reliably reproduce experimental results.

To mitigate this, we employed a before-and-after experimental design to compare test behavior before and after the implementation of specific interventions (e.g., RQ2). This approach allowed us to narrow and observe significant changes in flakiness patterns.

For RQ3, we further refined the approach by conducting separate benchmark runs for each DB sanity check and one combining the results of both sanity checks. This enabled us to attribute the observed changes to either one of the sanity checks and verify the behavior with the 'Both Dirty Groups Disabled' benchmark, providing a more precise understanding of the impact of each intervention.

Single-Snapshot Observations

All benchmarks were conducted on specific commits. This approach resulted in a limited representation of flaky tests. The observed flakiness patterns may not be representative of the full spectrum of flaky test failures encountered in the system over time. This is because the prevalence and characteristics of different types of flaky tests can vary significantly across various versions of the codebase. To illustrate, we observed that social interaction influenced the results: Prior to the RQ2 experiments, an engineer fixed 11 instances of implicit disposing that resulted in flaky tests based on our observations. In contrast, only 5 instances were resolved between benchmark runs. This demonstrates that the effectiveness of any interventions (e.g., RQ2) depends heavily on the specific types of flaky tests prevalent in the codebase at the time of the experiment. This limitation threatens the internal validity of the research, as the focus is on the possibility of certain types of flakiness rather than providing a distribution of certain flakiness.

Background Processes Overhead

Experiments were conducted on Sundays to minimize the impact of background processes running on the machines. This helped to ensure a more controlled and consistent experimental environment, benefiting the internal validity of the research. However, this may limit the ecological validity of the findings. To address this, we also investigated some runs during peak working hours and observed no significant differences.

Partial Cloud Migration

For RQ3, the system had undergone a partial migration to the cloud. This implies that some of the test runs were executed on different hardware configurations (on-premises versus cloud). These variations in hardware can introduce additional sources of variability in test execution times and potentially influence test flakiness. To minimize the impact on the internal validity of this research; We differentiated based on the platform where the tests were run and incorporated linked trends in the explanation of the individual results (Section 4.4.3).

Manually Resolving Flakiness

The manual resolution of prominent or easily identifiable flaky tests can introduce bias into the experimental results. Our examination focuses on the flaky state of the system at a specific point in time, and thus, we do not evaluate intermediate flaky states. Within these intermediate states, flakiness can both emerge and be resolved. Certain types of flakiness, such as those that are more problematic or easily debugged, are more likely to be addressed. This can result in a potentially skewed distribution of the remaining flakiness within the system, posing a threat to the internal validity of the research. However, this limitation more accurately reflects real-world development practices, thereby enhancing the ecological validity of the study.

Chapter 6

Conclusions and Future Work

This chapter concludes the thesis by summarizing the key findings and outlining potential avenues for future research. In Section 6.1, we present the primary conclusions drawn from our investigation into test flakiness within a large-scale industrial setting. We discuss the significant impact of flaky tests on software development and highlight the key findings of our research, including the identification and mitigation of various root causes of flakiness observed at Exact. In Section 6.2, we explore promising areas for future research, such as investigating the effectiveness of different types of flaky test information, refining methods for grouping flaky tests, extending the applicability of our tools, and delving deeper into the impact of mocking frameworks and singleton patterns on test flakiness.

6.1 Conclusions

Flaky tests pose a significant challenge in software development, hindering progress, wasting computational resources, obstructing automated repair mechanisms, and burdening developers with difficult-to-reproduce issues. This can ultimately erode trust in the testing process itself. While ignoring flakiness is not an option, as it can lead to increased software instability and potential failures, many organizations, such as Exact, struggle to find effective solutions. Despite implementing mitigation strategies, Exact still encountered over 100 failed test attempts in each Continuous Integration (CI) pipeline run.

This research investigated the root causes of flakiness within a large-scale industrial software system. The investigation comprised the following two key phases:

- **Characterizing the flaky test** landscape within Exact through a comprehensive analysis of flaky tests using a same-commit rerun approach.
- **Addressing the identified root causes of flakiness**, leading to the development of new tools and methodologies such as 'WeDispose', a code analysis rule for detecting and refactoring instances of the Implicit Dispose smell.

Through this case study at Exact we gained valuable insights into the various ways flakiness manifests in an industrial setting, including the impact of redundant database background tasks, dirty databases, and implicit resource disposal. Exact has significantly improved its test stability by addressing these issues; resulting in a record-high release rate.

Specifically, there was a notable reduction in flakiness after minimizing redundant database background tasks, with a 40% decrease in the average number of flaky tests observed per combined API and Integration CI run. Furthermore, by explicitly disposing of test data and disabling tests that leave the database in a dirty state, we respectively addressed the other 3% and 28% of the remaining flaky tests that on average failed more than 10% of their attempts. Most flakiness fixes were achieved through manual patches based on the flaky test reports generated throughout this study. All of these combined resulted in an increase of their Flakiness Pipeline Pass Percentage (FPPP), which indicates the chance of the CI pipeline passing with no changes, from 27% to 96%.

This research underscores the multifaceted nature of flakiness, influenced by a variety of factors, including external factors, increased flakiness resulting from platform changes, brittle tests, dirty tests, Implicit Dispose and other code smells, and the impact of database configurations. Through this work, we contribute the following:

- A novel same-commit rerunning approach to measuring test flakiness within a live industrial system.
- An illustration of the benefits of granular, grouped, and aggregated flaky test information to motivate organization members to address test flakiness.
- An illustration of the multifaceted and combinatorial nature of flakiness root causes and impacting factors in a database-reliant industrial system.

These findings are not limited to the specific context of Exact. They provide valuable insights and a framework for other organizations to address flakiness within their own software development processes. This includes illustrating flakiness behavior patterns, potential root causes with mitigation methods, and providing a framework for indexing their own flaky situation, including arguments and examples illustrating which information has proven useful and why.

By emphasizing the importance of finding and characterizing flaky tests, identifying their root causes, and implementing targeted mitigation strategies, this work aims to motivate both researchers and developers to start measuring and analyzing flaky tests. Enabling organizations to address flaky tests, thereby reducing server costs, improving software quality, accelerating development cycles, and enhancing overall developer productivity.

6.2 Future Work

This section outlines several promising avenues for future research inspired by our findings. We discuss the potential of a formal study on the impact of information provided to developers on flakiness reduction. We also explore the need for automated error message grouping and the extension of our tools to different testing frameworks. Finally, we propose further investigation into database dirtiness localization and the impact of Mocks, Stubs, and Singletons on test flakiness.

Empirical Study Into Flaky Information

We observed that the FPPP and release rate have increased since the onset of our research (Section 4.5). This improvement can be partially attributed to the sharing of flakiness status reports with developers (Section 5.1). While our research included informal feedback regarding the usefulness of this information, a formal investigation into the effects of different types of flaky information and their perceived usefulness would be valuable. To illustrate, within our work we observed that the FPPP and release rate have gone up since the start of our research (Section 4.5), partly due to the flakiness status reports shared with developers (Section 5.1). While we obtained informal feedback which information prove useful, formal and long-term evidence is missing. We therefore that a formal investigation regarding the effects of flaky information, and which information is useful, are the next steps towards practically addressing test flakiness. Future studies should therefore investigate the various aspects and benefits that flaky information provides, such as:

- The usefulness of the FPPP when structurally available in the development process.
- The potential benefits of replacing the flaky qualifier with a non-binary qualifier, such as a pass rate.
- The effectiveness of presenting tests based on flakiness, similar to the approach used to reduce test flakiness by half at Spotify [78].
- The benefits and overhead of generating flaky test reports based on reruns. The benefits can be quantified based on developer feedback or overall resource savings. Our median benchmark consisted of 26 pipeline runs, which consumed approximately 190 hours of compute resources without parallelization. This indicates a significant computational cost. However, during our research, which relied on this rerun benchmark approach, Exact executed 100 runs daily, resulting in an FPPP increase from 27% to 95%. This implies that by avoiding unnecessary rerunning of flaky pipelines, we saved 68 pipelines, or over 500[1] compute hours daily.

Grouping Flaky Tests

Grouping bug reports and their corresponding issues is a well-researched topic, with some work even applying categorization based on complex fault triggers [117]. Some research has focused on using machine learning to classify failing tests based on overlapping test alarms, although these approaches do not typically classify based on low-level error causes [48]. Other research has explored methods for automatically grouping bug reports [47]. Given the observed usefulness of manual grouping based on common phrases, we believe that a formal investigation into its effectiveness could be highly promising. Furthermore, this manual grouping method for bug reports can be extended to an automated grouping approach, similar to those described above, for broader applicability and easier implementation. This would enable an investigation into the optimal granularity level for grouping flaky tests.

[1] 500 hours is an estimation based on the average API and Integration test stage. Not every pipeline includes these stages; some may include more, while others may include fewer. They can include any subset of stages: Build (including unit tests), API, Integration, UI, and more.

Extending the Functionality of Our Tools

Our current tools do not support SpecFlow-generated tests. Zhang et al. [119] observed that automatically generated tests exhibit a higher frequency of order dependencies compared to manually written tests. Although SpecFlow-generated tests require human intervention, they still represent a level of abstraction that may reduce the consideration of data cleaning, potentially increasing the likelihood of order dependencies. To validate the sample validity and full applicability of our research, it would be valuable to evaluate how our tools perform with different languages or frameworks and with different databases, given that other research has identified differences in these areas [72, 36].

Other Database Dirtiness Approaches

Our current DB Sanity Check offers a broad, proactive approach to addressing flakiness caused by data leaks. However, its functionality can be extended to handle dirtiness with statement precision and enabling the detection of dirtiness introduced by test classes or assemblies (Section 5.3.2). We observed that 3% of the Implicit Dispose smell stemmed from the class level. While prior work suggests that most dirtiness stems from individual test code [37], based on our observations, we believe class-level dirtiness might contribute to a significant number of flaky tests. Furthermore, our sanity checks reported over 11% of tests with unclear culprits. Being able to pinpoint dirtiness to the specific statement or data-polluting query would address these two issues and might therefore be more effective. We believe a dynamic approach similar to Huo et al. [46] adjusted for databases – possibly by utilizing the static *Database Interaction Control Flow Graph* (DICFG) information from Kapfhammer et al. [50] instead of a normal CFG – might be more effective in this context. This approach could be extended by incorporating an execution trace comparison method similar to Ziftci et al. [72] to enable the identification of specific database inter-test dependencies that contribute to changes in the behavior of flaky tests.

Impact of Mocks, Stubs and Singletons on Test Flakiness

As stated in Section 4.3.2, a significant portion of unfazed non-sporadic flaky tests originated from stubs persisting between tests through singletons. While other research has investigated the impact of static variables [119], it does not specifically consider static variables such as these singletons, where the default singleton is allowed to persist, but stubs are not. Based on our observation that a significant portion of flakiness at Exact stems from such issues, further investigation into this area could be highly valuable. This can be achieved by adapting PolDet [37] to account for .NET languages and to differentiate between Mocks/Stubs and normal functions. Alternatively, a more readily applicable solution could involve addressing this issue similarly to the Implict Dispose smell, either by designing a new Roslyn rule or by adapting the stubs such that they can utilize `using` or the WeDispose rule. This adaptation would involve changing the type of the Stub inserted into a singleton to inherit from an `IDisposable`; A refactor founded on the principle that a singleton, within the context of a test class, represents unmanaged data that needs to be cleaned.

Bibliography

[1] Abdullahi Abubakar Imam, Shuib Basri, Rohiza Ahmad, and María T. González-Aparicio. Literature review on database design testing techniques. In Radek Silhavy, editor, *Software Engineering Methods in Intelligent Algorithms*, pages 1–13, Cham, 2019. Springer International Publishing. ISBN 978-3-030-19807-7. URL https://doi.org/10.1007/978-3-030-19807-7_1.

[2] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 83–93, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2771786. URL https://dl.acm.org/doi/abs/10.1145/2771783.2771786.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 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*, pages 185–196, 2024. URL https://arxiv.org/abs/2402.09171.

[4] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. FlakeFlagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1572–1584, May 2021. doi: 10.1109/ICSE43902.2021.00140. URL https://ieeexplore.ieee.org/abstract/document/9402098/.

[5] Andomar. Rollback the inner transaction of nested transaction, 2016. URL https://stackoverflow.com/questions/12456579/rollback-the-inner-transaction-of-nested-transaction. Accessed 30 Jan 2025.

[6] Ali Asad, Hamza Ali, Ali Asad, and Hamza Ali. Manage object life cycle. *The C# Programmer's Study Guide (MCSD) Exam: 70-483*, pages 197–205, 2017.

[7] Kent Beck. Facebook engineering process with Kent Beck. Software Engineering Daily, 2019. URL https://softwareengineeringdaily.com/2019/08/28/facebook-engineering-process-with-kent-beck/. Accessed 30 Jan 2025.

[8] Boris Beizer. *Software testing techniques, Chapter Two*. International Thomson Computer Press, 1990.

[9] Jonathan Bell and Gail Kaiser. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 550–561, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568248. URL https://dl.acm.org/doi/abs/10.1145/2568225.2568248.

[10] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 770–781, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786823. URL https://dl.acm.org/doi/abs/10.1145/2786805.2786823.

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

[12] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939951. doi: 10.1145/1453101.1453146. URL https://dl.acm.org/doi/abs/10.1145/1453101.1453146.

[13] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and 3 Sunghun Kim. Duplicate bug reports considered harmful ... really? In *2008 IEEE International Conference on Software Maintenance*, pages 337–345, Sep. 2008. doi: 10.1109/ICSM.2008.4658082. URL https://ieeexplore.ieee.org/abstract/document/4658082/.

[14] Megan Bradley, Masha Thomas, Randolph West, Jason Roth, Niko Neugebauer, Mike Ray, Craig Guyer, and Douglas Laudenschlager. Rollback work (Transact-SQL), 2024. URL https://learn.microsoft.com/en-us/sql/t-sql/language-elements/rollback-work-transact-sql?view=sql-server-ver16. Microsoft Official .NET Docs. Accessed 30 Jan 2025.

[15] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 14(1):17–44, 2004. doi: https://doi.org/10.1002/stvr.286. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.286.

[16] Andy Chou. Static analysis in industry, 2014. URL https://popl.mpi-sws.org/2014/andy.pdf. Coverity. Accessed 30 Jan 2025.

[17] The Business Research Company. Database software global market report 2025, 2025. URL https://www.thebusinessresearchcompany.com/report/database-software-global-market-report. Accessed 30 Jan 2025.

[18] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, second edition, 01 2005. ISBN 9780321578815.

[19] Hugo Da Gião, André Flores, Rui Pereira, and Jácome Cunha. Chronicles of CI/CD: A deep dive into its usage over time. *arXiv preprint arXiv:2402.17588*, 2024.

[20] Aswath Damodaran. Historical (compounded annual) growth rates by sector, 2024. URL https://pages.stern.nyu.edu/~adamodar/New_Home_Page/datafile/histgr.html. Accessed 30 Jan 2025.

[21] Arie Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, CWI, 2001. URL https://www.researchgate.net/publication/2534882_Refactoring_Test_Code.

[22] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. Concurrency-related flaky test detection in Android apps. *CoRR*, abs/2005.10762, 2020. URL https://arxiv.org/abs/2005.10762.

[23] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 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*, ESEC/FSE 2019, page 830–840, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338945. URL https://dl.acm.org/doi/abs/10.1145/3338906.3338945.

[24] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003. doi: https://doi.org/10.1002/cpe.654. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.654.

[25] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 151–162, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937346. doi: 10.1145/1273463.1273484. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1273463.1273484.

[26] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010. ISSN

0950-5849. doi: https://doi.org/10.1016/j.infsof.2009.07.001. URL https://www.sciencedirect.com/science/article/pii/S0950584909001219.

[27] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! Characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 62–71, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597098. URL https://dl.acm.org/doi/abs/10.1145/2597073.2597098.

[28] Martin Fowler. Eradicating non-determinism in tests, 2011. URL https://martinfowler.com/articles/nonDeterminism.html. Accessed 30 Jan 2025.

[29] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11, April 2018. doi: 10.1109/ICST.2018.00011. URL https://ieeexplore.ieee.org/abstract/document/8367031/.

[30] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 55–65, May 2015. doi: 10.1109/ICSE.2015.28. URL https://ieeexplore.ieee.org/abstract/document/7194561/.

[31] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.12.013. URL https://www.sciencedirect.com/science/article/pii/S0164121217303060.

[32] Gartner and Statista. Size of the database management system (DBMS) market worldwide from 2017 to 2021, 2024. URL https://www.statista.com/statistics/724611/worldwide-database-market/. Accessed 30 Jan 2025.

[33] L. George and P. Minet. A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 441–448, 1997. doi: 10.1109/ICDCS.1997.603275. URL https://ieeexplore.ieee.org/abstract/document/603275/.

[34] Maxime Gobert, Csaba Nagy, Henrique Rocha, Serge Demeyer, and Anthony Cleve. Best practices of testing database manipulation code. *Information Systems*, 111:102105, 2023. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2022.102105. URL https://www.sciencedirect.com/science/article/pii/S0306437922000886.

[35] Google. TotT: Avoiding flakey tests, 2008. URL https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html. Accessed 30 Jan 2025.

[36] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of flaky tests in Python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 148–158, April 2021. doi: 10.1109/ICST49551.2021.00026. URL https://ieeexplore.ieee.org/abstract/document/9438576/.

[37] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 223–233, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2771793. URL https://dl.acm.org/doi/abs/10.1145/2771783.2771793.

[38] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 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*, FSE 2016, page 993–997, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2983932. URL https://dl.acm.org/doi/abs/10.1145/2950290.2983932.

[39] Mark Harman and Peter O'Hearn. 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)*, pages 1–23, Sep. 2018. doi: 10.1109/SCAM.2018.00009. URL https://ieeexplore.ieee.org/abstract/document/8530713/.

[40] Sam Harwell. Improve performance of CA2000 (DisposeObjectsBeforeLosingScope), 2021. URL https://github.com/dotnet/roslyn-analyzers/issues/4915. Accessed 30 Jan 2025.

[41] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. An empirical study of flaky tests in JavaScript. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34, Oct 2022. doi: 10.1109/ICSME55016.2022.00011. URL https://ieeexplore.ieee.org/abstract/document/9978194/.

[42] Marcus Heege. Reliable resource management. *Expert C++/CLI: . NET for Visual C++ Programmers*, pages 253–278, 2007.

[43] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48, May 2015. doi: 10.1109/ICSE.2015.133. URL https://ieeexplore.ieee.org/abstract/document/7202948/.

[44] Itti Hooda and Rajender Singh Chhillar. Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13), 2015. URL https://research.ijcaonline.org/volume111/number13/pxc3901433.pdf.

[45] Zoran Horvat. From Dispose pattern to auto-disposable objects in .NET, 2019. URL https://codinghelmet.com/articles/from-dispose-pattern-to-auto-disposable-objects-in-net. Accessed 30 Jan 2025.

[46] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 621–631, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635917. URL https://dl.acm.org/doi/abs/10.1145/2635868.2635917.

[47] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 52–61, June 2008. doi: 10.1109/DSN.2008.4630070. URL https://ieeexplore.ieee.org/abstract/document/4630070/.

[48] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. What causes my test alarm? automatic cause analysis for test alarms in system and integration testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 712–723, May 2017. doi: 10.1109/ICSE.2017.71.

[49] Yunfeng Jiang and Yijun Jiang. Analysis engine for automatically analyzing and linking error logs, 2016. URL https://patents.google.com/patent/US9424115B2/en. US Patent 9.424,115 B2.

[50] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, page 98–107, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137435. doi: 10.1145/940071.940086. URL https://dl.acm.org/doi/abs/10.1145/940071.940086.

[51] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, page 1–11, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934685. doi: 10.1145/1181775.1181777. URL https://dl.acm.org/doi/abs/10.1145/1181775.1181777.

[52] Kirsan. Flow analysis for CA2000 (DisposeObjectsBeforeLosingScope) consumes overwhelming resources, 2023. URL https://github.com/dotnet/roslyn-analyzers/issues/6412. Accessed 30 Jan 2025.

[53] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. Modeling and ranking flaky tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*,

ICSE-SEIP '20, page 110–119, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371230. doi: 10.1145/3377813.3381370. URL https://dl.acm.org/doi/abs/10.1145/3377813.3381370.

[54] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 821–830, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106288. URL https://dl.acm.org/doi/abs/10.1145/3106237.3106288.

[55] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 101–111, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330570. URL https://dl.acm.org/doi/abs/10.1145/3293882.3330570.

[56] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, April 2019. doi: 10.1109/ICST.2019.00038. URL https://ieeexplore.ieee.org/abstract/document/8730188/.

[57] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1471–1482, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3381749. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3377811.3381749.

[58] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–413, Oct 2020. doi: 10.1109/ISSRE5003.2020.00045. URL https://ieeexplore.ieee.org/abstract/document/9251071/.

[59] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428270. URL https://dl.acm.org/doi/abs/10.1145/3428270.

[60] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page

1381–1392, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473931. URL https://dl.acm.org/doi/abs/10.1145/3468264.3473931.

[61] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. Effectively manifesting concurrency bugs in Android apps. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 209–216, Dec 2016. doi: 10.1109/APSEC.2016.038. URL https://ieeexplore.ieee.org/abstract/document/7890590/.

[62] Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto. Unsupervised bug report categorization using clustering and labeling algorithm. *International Journal of Software Engineering and Knowledge Engineering*, 26(07):1027–1053, 2016. doi: 10.1142/S0218194016500352. URL https://doi.org/10.1142/S0218194016500352.

[63] Jeff Listfield. Where do our flaky tests come from?, 2017. URL https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html. Accessed 30 Jan 2025.

[64] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635920. URL https://dl.acm.org/doi/abs/10.1145/2635868.2635920.

[65] Mateusz Machalica, Wojtek Chmiel, Swierc Stanislaw, and Ruslan Sakevych. How do you test your tests?, 2020. URL https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/. Engineering at Meta. Accessed 30 Jan 2025.

[66] Jean Malm, Adnan Causevic, Björn Lisper, and Sigrid Eldh. Automated analysis of flakiness-mitigating delays. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, AST '20, page 81–84, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379571. doi: 10.1145/3387903.3389320. URL https://dl.acm.org/doi/abs/10.1145/3387903.3389320.

[67] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the Defects4j dataset. *Empirical Software Engineering*, 22:1936–1964, 2017. URL https://doi.org/10.1007/s10664-016-9470-4.

[68] Nočnica Mellifera. How to diagnose flaky tests, 2023. URL https://thenewstack.io/how-to-diagnose-flaky-tests/. Accessed 30 Jan 2025.

[69] Nočnica Mellifera. Articles on Medium tagged with 'Flaky Tests', 2025. URL https://medium.com/tag/flaky-tests. Accessed 30 Jan 2025.

[70] John Micco. Flaky tests at Google and how we mitigate them, 2016. URL https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed 30 Jan 2025.

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

[72] John Micco and Atif Memon. How flaky tests in continuous integration, 2016. URL https://www.youtube.com/watch?v=CrzpkF1-VsA. Google Test Automation Conference 2016.

[73] Jesús Morán, Cristian Augusto, Antonia Bertolino, Claudio De La Riva, and Javier Tuya. FlakyLoc: Flakiness localization for reliable test suites in web applications. *Journal of Web Engineering*, 19(2):267–296, March 2020. ISSN 1544-5976. doi: 10.13052/jwe1540-9589.1927. URL https://ieeexplore.ieee.org/abstract/document/10247298/.

[74] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 496–499, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025202. URL https://dl.acm.org/doi/abs/10.1145/2025113.2025202.

[75] G Murphy and Davor Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 1–6. Citeseer, 2004. URL http://www.cs.ubc.ca/labs/spl/papers/2004/seke04-bugzilla.pdf.

[76] Robert O'Callahan. Introducing Chaos Mode, 2014. URL https://robert.ocallahan.org/2014/03/introducing-chaos-mode.html. Accessed 30 Jan 2025.

[77] Stack Overflow. Stack Overflow post tagged with 'idisposable', 2025. URL https://stackoverflow.com/questions/tagged/idisposable?sort=MostVotes. Accessed 30 Jan 2025.

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

[79] Owain Parry. *Understanding and Mitigating Flaky Software Test Cases*. PhD thesis, University of Sheffield, 2023.

[80] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. Flake it 'till you make it: Using automated repair to induce and fix latent test flakiness. In

Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20, page 11–12, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379632. doi: 10.1145/3387940.3392177. URL https://dl.acm.org/doi/abs/10.1145/3387940.3392177.

[81] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol.*, 31(1), October 2021. ISSN 1049-331X. doi: 10.1145/3476105. URL https://dl.acm.org/doi/abs/10.1145/3476105.

[82] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(3):72, 2023.

[83] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 492–502, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi: 10.1145/3379597.3387482. URL https://dl.acm.org/doi/abs/10.1145/3379597.3387482.

[84] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. Wait, wait. No, tell me. Analyzing Selenium configuration effects on test flakiness. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 7–13, May 2019. doi: 10.1109/AST.2019.000-1. URL https://ieeexplore.ieee.org/abstract/document/8821891/.

[85] Md Tajmilur Rahman and Peter C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 857–862, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3275529. URL https://dl.acm.org/doi/abs/10.1145/3236024.3275529.

[86] Maaz Hafeez Ur Rehman and Peter C. Rigby. 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*, ESEC/FSE 2021, page 1371–1380, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473930. URL https://dl.acm.org/doi/abs/10.1145/3468264.3473930.

[87] Michael Robinson. Test failure bucketing, 2014. URL https://patents.google.com/patent/US8782609B2/en. US Patent 8,782,609 B2.

[88] Jakub Rumpel. Be careful using ROLLBACK on nested transaction in SQL Server!, 2021. URL https://dev.to/hanewali/be-careful-using-rollback-on-nested-transaction-in-sql-server-2b5p. Accessed 30 Jan 2025.

[89] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 499–510, May 2007. doi: 10.1109/ICSE.2007.32. URL https://ieeexplore.ieee.org/abstract/document/4222611/.

[90] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[91] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, April 2016. doi: 10.1109/ICST.2016.40. URL https://ieeexplore.ieee.org/abstract/document/7515461/.

[92] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 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*, ESEC/FSE 2019, page 545–555, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338925. URL https://dl.acm.org/doi/abs/10.1145/3338906.3338925.

[93] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. Shake it! Detecting flaky tests caused by concurrency with Shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311, Sep. 2020. doi: 10.1109/ICSME46990.2020.00037. URL https://ieeexplore.ieee.org/abstract/document/9240694/.

[94] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. The effects of computational resources on flaky tests. *IEEE Transactions on Software Engineering*, 50(12):3104–3121, Dec 2024. ISSN 1939-3520. doi: 10.1109/TSE.2024.3462251. URL https://ieeexplore.ieee.org/abstract/document/10682606/.

[95] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. To mock or not to mock? An empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 402–412, May 2017. doi: 10.1109/MSR.2017.61. URL https://ieeexplore.ieee.org/abstract/document/7962389/.

[96] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. Testing Scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 165–175, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338910. URL https://dl.acm.org/doi/abs/10.1145/3338906.3338910.

[97] Statista. Software - Worldwide, 2024. URL https://www.statista.com/outlook/tmo/software/worldwide. Accessed 30 Jan 2025.

[98] Pavan Sudarshan. No more flaky tests on the Go team, 2012. URL https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team. Accessed 30 Jan 2025.

[99] Amjed Tahir, Shawn Rasheed, Jens Dietrich, Negar Hashemi, and Lu Zhang. Test flakiness' causes, detection, impact and responses: A multivocal review. *Journal of Systems and Software*, 206:111837, 2023. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2023.111837. URL https://www.sciencedirect.com/science/article/pii/S0164121223002327.

[100] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 69–72, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371261. doi: 10.1145/3377816.3381742. URL https://dl.acm.org/doi/abs/10.1145/3377816.3381742.

[101] Swapna Thorve, Chandani Sreshtha, and Na Meng. An empirical study of flaky tests in Android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538, Sep. 2018. doi: 10.1109/ICSME.2018.00062. URL https://ieeexplore.ieee.org/abstract/document/8530060/.

[102] Google Trends. 'flaky test' search term trend, 2024. URL https://trends.google.com/trends/explore?date=all&q=flaky%20test,flakey%20test,flaky%20tests. Accessed 30 Jan 2025.

[103] Edward Triou Jr, Andre Milbradt, Osarumwemse U Agbonile, and Affan Arshad Dar. Systems and methods for automated classification and analysis of large volumes of test result data, 2009. URL https://patents.google.com/patent/US6424971B1/en. US Patent 7,509,538.

[104] WeiTek Tsai, XiaoYing Bai, and Yu Huang. Software-as-a-service (SaaS): perspectives and challenges. *Science China Information Sciences*, 57:1–15, 2014. doi: 10.1007/s11432-013-5050-z. URL https://dl.acm.org/doi/abs/10.1145/2635868.2635920.

[105] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(75)80008-0. URL https://www.sciencedirect.com/science/article/pii/S0022000075800080.

[106] Lionel Vailshery. Database software - statistics & facts, 2024. URL https://www.statista.com/topics/5157/database-software/. Accessed 30 Jan 2025.

[107] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 327–337, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409709. URL https://dl.acm.org/doi/abs/10.1145/3368089.3409709.

[108] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access*, 9:76119–76134, 2021. ISSN 2169-3536. doi: 10.1109/ACCESS.2021.3082424. URL https://ieeexplore.ieee.org/abstract/document/9437181/.

[109] Roger Villela and Roger Villela. Unmanaged .NET data types and system. io. *Understanding System. IO for. NET Core 3: Implementing Internal and Commercial Tools*, pages 153–169, 2020.

[110] Ruixin Wang, Yang Chen, and Wing Lam. 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*, ICSE '22, page 120–124, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392235. doi: 10.1145/3510454.3516846. URL https://dl.acm.org/doi/abs/10.1145/3510454.3516846.

[111] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 461–470, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368151. URL https://dl.acm.org/doi/abs/10.1145/1368088.1368151.

[112] Genevieve Warren, Fredi Kats, Petr Kulikov, Stefan Seeland, Travis Stewart, Amaury Leve, Youssef Victor, Manish Vasani, and David Coulter. CA2000: Dispose objects before losing scope, 2024. URL https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2000. Microsoft Official .NET Docs. Accessed 30 Jan 2025.

[113] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. Preempting flaky tests via non-idempotent-outcome tests. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1730–1742, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510170. URL https://dl.acm.org/doi/abs/10.1145/3510003.3510170.

[114] D. Willmor and S.M. Embury. A safe regression test selection technique for database-driven applications. In *21st IEEE International Conference on Software Maintenance*

(ICSM'05), pages 421–430, Sep. 2005. doi: 10.1109/ICSM.2005.15. URL https://ieeexplore.ieee.org/abstract/document/1510137/.

[115] Dean C Wills and Dean C Wills. Final thoughts. *C++ 2013 for C# Developers*, pages 335–352, 2014.

[116] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in software engineering*, volume 236. Springer, 2012.

[117] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Automatic defect categorization based on fault triggering conditions. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 39–48, Aug 2014. doi: 10.1109/ICECCS.2014.14. URL https://ieeexplore.ieee.org/abstract/document/6923116/.

[118] Lei Zhang, Mahsa Radnejad, and Andriy Miranskyy. Identifying flakiness in quantum programs. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–7. IEEE, 2023. doi: 10.1109/ESEM56168.2023.10304850. URL https://ieeexplore.ieee.org/abstract/document/10304850/.

[119] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 385–396, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2610404. URL https://dl.acm.org/doi/abs/10.1145/2610384.2610404.

[120] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. Research progress of flaky tests. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 639–646, March 2021. doi: 10.1109/SANER50967.2021.00081. URL https://ieeexplore.ieee.org/abstract/document/9425992/.

[121] Celal Ziftci and Diego Cavalcanti. 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)*, pages 736–745, 2020. doi: 10.1109/ICSME46990.2020.00083. URL https://ieeexplore.ieee.org/abstract/document/9240685/.

[122] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, Sep. 2010. ISSN 1939-3520. doi: 10.1109/TSE.2010.63. URL https://ieeexplore.ieee.org/abstract/document/5487527/.

Appendix A

Glossary

In the Appendix we will extensively explain some of the methodology used within this thesis. Furthermore in the next section (Appendix A.2) we show an observation of our FPPP and its binding constrains with separate test stages.

A.1 Terminology

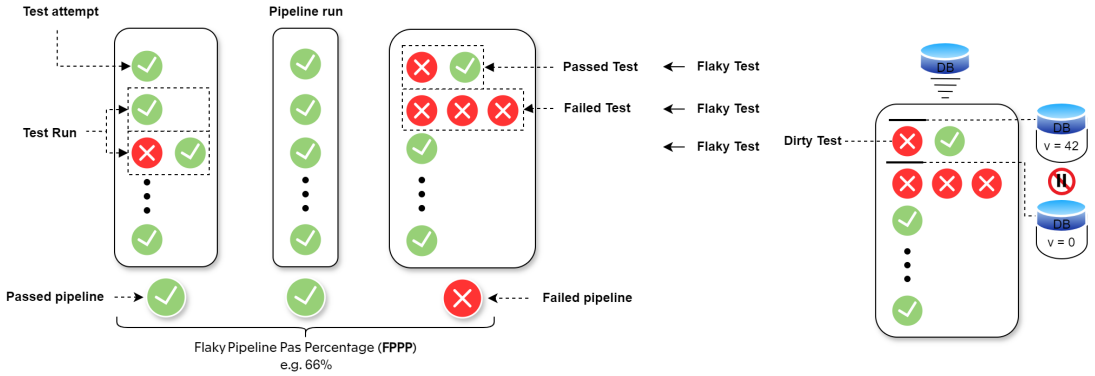
This section defines terminology that is used throughout the thesis. Some terminology is unique to this paper such as 'Flaky Pipeline Pass Percentage' (FPPP) while other terms such as 'Flaky tests' are explicitly stated because various definitions are used within the academic world. Figure A.1 provides a visual representation of all definitions.

First and foremost every time we mention Exact we only reference their main product Exact Online and its coherent departments. Furthermore a **test attempt** is a single execution of the test code. It will either 'fail' or 'pass'. A collection of test attempts for one test within a pipeline run is called a **test run** or an execution of a test. A test run executes one or more times until it either passes or fails the maximum number of attempts. Within Exact this maximum number of attempts is 3 times. The aforementioned **pipeline run** is an execution of all active tests in the entire Test Suite. The pipeline run can either pass, if all tests pass, or fail if at least one test fails. Within Exact the **pipeline** consists of all active Integration, UI and API tests, with its test and production code. The pipeline is linked to a commit to specify the specific version of the source code. Note that this is a many to one relation, and that within this research we do multiple pipeline runs on a single commit to uncover flaky tests.

The reason why two test attempts can differ within a test run or over multiple pipeline runs over the same commit is because of flaky tests. A test is considered flaky for a commit (**flaky test**) if it has at least one failed and one passing attempt in all its test runs for all its pipeline runs for that commit. Note that a flaky test is not necessarily an incorrect test. We consider an **incorrect test** as a test that causes unwanted behavior. These definitions intersect but neither is a subset of the other. Flakiness in a correct test can be caused by another (in)correct test. Think for example of two test that require a resource and the flaky test times out because the other tests holds the lock to the resource. Another example of how a non-flaky incorrect test can cause a correct test to be flaky, is through manipulation of persistent data in the test environment.

We call tests (such as these) that leave the test environment in a different state after execution compared to pre-execution state, a **Dirty test**.

We define a new subcategory of flaky test within this thesis, a **first attempt failure** test. This is a test that consistently fails on its first attempt, and can pass on a subsequent attempts. To make the impact of test flakiness more tangible we define the metric **Flaky Pipeline Pas Percentage (FPPP)**. This describes the chance of a pipeline run passing for a certain commit. We defined this metric as it describes an important negative direct result from flakiness in your pipeline.



(a) Definitions in an experiment

(b) Definitions of a dirty test

Figure A.1: Visual representation of definitions

A.2 Independent FPPP's

The flaky stages are executed in parallel and therefore allow for Independent FPPP's. The CI pipeline is set up in multiple stages, where it first runs all steps sequentially and then finally runs the API, Integration, and optionally UI tests in their own parallel executed stage. This prevents an order dependency within flaky stages such that when one stage fails, the other stage still executes. This means one can calculate an independent FPPP for all flaky test stages such as API and Integration tests. These independent FPPP's do not allow a calculation of the overall FPPP but do allow us to bind it accordingly: $\prod FPPP_s \leq FPPP_{overall} \leq \min FPPP_s, \forall s$, where s is a parallel executed stage. This independent FPPP provides a clearer localization of the flakiness problem.