ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Integrating flakiness detection and repair

Alexandre Filipe de Freitas Mendes

A dissertation submitted in partial fulfillment of
the requirements for the degree of Master in Informatics Engineering
Specialization Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, July 2024

Acknowledgments

I would like to express my gratitude to my advisor, Isabel Azevedo, whose guidance and understanding were invaluable to the completion of this study.

A special thanks to my colleagues and friends, Tiago Barbosa, Rogério Alves, João Ribeiro, and Pedro Marques, for their support and the fun we had throughout these last few years.

Finally, I would also like to thank my parents, sister, and closest friends for their support and encouragement that enabled me to carry on when times were harder. This accomplishment would not have been possible without their support.

Thank you all.

Declaration of Integrity

I declare myself to have conducted this academic work with integrity.

I have not plagiarized or applied any form of improper use of information or falsification of results throughout the process that led to its preparation.

Therefore, the work presented in this document is original and my own and has not previously been used for any purpose not related to this project in its different phases. I also declare that I am fully aware of the Code of Ethical Conduct of P. PORTO.

ISEP, Porto, 4 July 2024

Abstract

Testing is a core component of software development, safeguarding the reliability of the code. However, some tests may provide unexpected outcomes, non-deterministically passing or failing. Such tests are deemed flaky. Known to delay releases and reduce the overall effectiveness and efficiency of testing these tests have become a key problem. Affecting the entire industry, awareness, and research on the topic have risen significantly in recent years.

This dissertation presents a systematic literature review on flaky test research's state of the art. It details the research methodology used and provides an overview of recent developments in the field. The review addresses four dimensions: causes, detection, repair, and tool integration. Each dimension is analyzed to classify flakiness and detail tools and methodologies capable of addressing different aspects of flakiness. In this manner, the dissertation comprehensively examines recent strategies for addressing flaky tests.

Among the categories of flakiness, order-dependency has received the most research focus, with several tools available for detecting and repairing this category specifically. However, there is a notable lack of tools capable of both detecting and repairing flaky tests in a unified manner. By studying and evaluating individual tools, this dissertation assesses the potential compatibility between already existing and available methodologies, allowing for the development of a prototype for a unified order-dependency flakiness solution.

Evaluating this prototype against the individual execution of its components reveals that integrating existing tools enhances usability while preserving comparable levels of recall and execution time. This analysis highlights the necessity for standardizing the inputs and outputs of order-dependency detection and repair tools. Additionally, it shows that complete detection and repair solutions can be effectively developed using pre-existing components as a basis.

Keywords: Flaky tests, Non-determinism, Software testing, Order dependency

Resumo

Os testes são um componente essencial do desenvolvimento de software, garantindo a confiabilidade do código. No entanto, alguns testes podem fornecer resultados inesperados, passando ou falhando de maneira não determinística. Estes testes são chamados flaky. Conhecidos por atrasar lançamentos e reduzir a eficácia e eficiência geral dos testes, esses testes tornaram-se um problema importante. Afetando toda a indústria, a consciencialização e a pesquisa sobre o tema aumentaram significativamente nos últimos anos.

Esta dissertação apresenta um panorama geral sobre o estado da arte da pesquisa de testes flaky realizando uma revisão sistemática da literatura. Detalha a metodologia de pesquisa utilizada e fornece uma visão geral dos desenvolvimentos recentes na área. A revisão aborda quatro dimensões: causas, detecção, reparo e integração de ferramentas. Cada dimensão é analisada para classificar a flakiness e detalhar ferramentas e metodologias capazes de abordar diferentes aspectos da mesma. Desta forma, a dissertação examina de forma abrangente as estratégias recentes para lidar com testes flaky.

Entre as categorias de flakiness, a dependência de ordem recebeu o maior foco de pesquisa, com várias ferramentas disponíveis para detectar e reparar especificamente essa categoria. No entanto, há uma falta notável de ferramentas capazes de detectar e reparar testes instáveis de maneira unificada. Ao estudar e avaliar ferramentas individuais, esta dissertação avalia a compatibilidade potencial entre metodologias existentes, abrindo caminho para o desenvolvimento de um protótipo para uma solução unificada de instabilidade por dependência de ordem.

A avaliação deste protótipo em comparação com a execução individual dos seus componentes revela que a integração de ferramentas pré-existentes melhora a usabilidade, mantendo níveis comparáveis de recall e tempo de execução. Esta análise destaca a necessidade de padronizar as entradas e saídas de ferramentas de deteção e reparação de dependência de ordem. Adicionalmente, mostra que soluções completas de deteção e reparação podem ser desenvolvidas eficazmente utilizando componentes pré-existentes como base.

Palavras-chave: Testes flaky, Não-determinismo, Testes de software, Dependência de ordem

Table of Contents

List of Figures

List of Tables

Abbreviations and Symbols

CI	Continuous Integration
C_i*	Relative closeness to the ideal solution
CUT	Code under test
GQM	Goal Question Metric
ML	Machine learning
NOD	Non-order-dependent
OD	Order-dependent
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
RTS	Regression test selection
SLR	Systematic Literature Review
TOPSIS	Technique for Order of Preference by Similarity to Ideal Solution
UI	User Interface
VM	Virtual Machine

1 Introduction

This chapter provides context and articulates the problem under examination, delineates the objectives of the study, elucidates the research methodology employed, and outlines the structure of the document.

1.1 Context

Testing is an integral part of software development, ensuring the reliability, functionality, and performance of software systems. In the context of modern software development practices, characterized by continuous integration and delivery, the efficiency of testing processes becomes increasingly critical.

When tests fail developers typically debug their local changes to identify and correct the root cause. Sometimes, however, the test result is unexpectedly non-deterministic leading to difficulty or even inability to find the root cause. Coined as 'flaky', these tests present a challenge to the dependability of test suites, its causes being varied and widespread.

This issue affects companies and institutions across the industry. Noteworthy studies, such as the one conducted by Google, highlight that up to 80% of failing tests can be attributed to flakiness (Leong *et al.*, 2019). This issue is further underscored by an examination of 211 projects within the Apache Software Foundation, revealing that 21% of false alarms (false fail results) stem from flaky tests (Vahabzadeh, Fard and Mesbah, 2015). Additionally, a recent developer survey underscores the prevalence of this issue, with 56% of respondents claiming to contend with flaky tests on a monthly, weekly, or daily basis (Parry, Kapfhammer, *et al.*, 2022).

Given the widespread impact of flaky tests, a need for solutions has emerged. However, current methods proposed by industrial teams and research institutes for detecting flakiness fall short in terms of reliability, efficiency, and resource utilization. These approaches also contribute to an increased turnaround time within the developer feedback loop (Zolfaghari *et al.*, 2021).

1.2 Problem

The definition of what constitutes a flaky test varies from source to source but is generally considered to mean a test that can be observed to both pass and fail without changes to the code under test (Parry *et al.*, 2021).

Flaky tests significantly impact software testing techniques since these rely on assumptions of deterministic outcomes and test independence. Product quality is impacted due to the unreliability flakiness introduces to the CI workflow, affecting software maintenance for techniques such as test-based program repair, crash reproduction, test amplification, and fault localization (Tahir *et al.*, 2023). The disruptions extend to human aspects, harming developer productivity and confidence in testing (Parry, Kapfhammer, *et al.*, 2022), as revealed in a survey on developer impact (Gruber and Fraser, 2022), where more than 60% of participants reported delayed releases due to flakiness.

In response, the software development community has been seeking solutions. The most common method of detecting flakiness centers around test repetition, as seen in the testing infrastructure of major companies like Google (Micco, 2017) and Microsoft (Lam, Muslu, *et al.*, 2020). This approach involves subjecting a test to the same Code Under Test (CUT) with the same test code a given number of times. A variance in results across these repetitions is indicative of flakiness in the test (Zolfaghari *et al.*, 2021). Despite its unreliability and cost, this approach is ubiquitous due to its simplicity and the ability to alter the number of re-runs to suit the project's needs. A study conducted by (Alshammari *et al.*, 2021) found that, out of 10,000 runs, only roughly a quarter of identified flaky tests would have been detected with just 10 reruns, underscoring the limitations of this simple detection approach.

While research primarily focuses on the detection of flaky tests, their repair is equally important. Manual fixing proves ineffective, as developers frequently claim to fix a flaky test without reducing the frequency of failures (Lam, Muslu, *et al.*, 2020). The fixing process is time-consuming, with developers taking around 18 days to resolve flaky bug reports, compared to 13 days for non-flaky issues (Lam, Muslu, *et al.*, 2020).

This underscores the need for efficient tools capable of identifying and subsequently repairing flaky tests, mitigating the time, resource, and human efforts necessary to address this pervasive issue. Presently, a noticeable gap exists between the availability of tools capable of detecting or repairing flakiness and tools that can automatically perform both subsequently. As a result, developers are left with limited practical solutions to apply to real life projects since the ones available require significant manual effort for either detection or repair.

1.3 Objective and Research Methodology

The global objective of this research is to explore approaches to detect and correct flaky tests and contribute to the field. To that end, the goal is to propose and assess the viability of an approach that unifies available detection and repair components to automatically address order-dependency flakiness. With this goal in mind, these relevant research questions were defined:

- RQ1. What are the causes and associated factors of flaky tests? (detailed in Section 3.2.2)

- RQ2. What approaches are available for detecting flaky tests? (detailed in Section 3.2.3)

- RQ3. What approaches are available for repairing flaky tests? (detailed in Section 3.2.4)

- RQ4. What are the benefits and drawbacks of a unified approach for detecting and repairing flakiness caused by order-dependent tests? (detailed in Section 3.2.5)

In pursuit of the established goal, a Systematic Literature Review (SLR) was performed since it provides a systematic and reproducible method for identifying, selecting, and appraising studies relevant to a specific research question. Employing this method ensures a comprehensive research synthesis, offering a reliable foundation for addressing the targeted review questions (MacDonald, 2014). The SLR, detailed in Chapter 3, explores the defined research questions and builds a knowledge foundation to allow further analysis of the subject.

With the insights from the SLR, RQ4 remains partially explored. To answer this question the Goal Question Metric (GQM) (Basili, Caldiera and Rombach, 1994) approach is used to assist in performing a controlled experiment. A GQM model is a hierarchical structure where a goal is refined into several measurement questions that break down the issue into its major components. Each question is then refined into metrics capable of sustaining the answer. Considering the outlined goal, the measurement questions (MQ) and their corresponding metrics were detailed in Table 1. Table 30 in Appendix A provides an extended description of the metrics.

Following the SLR and with the defined goal in mind, an empirical comparative analysis of multiple tools and methodologies based on the defined metrics is performed. This evaluation is part of the controlled experiment and contemplates 10 projects totaling 444 flaky tests. It includes both detection and repair techniques and forms the basis for the design and implementation of a unified solution.

Subsequently, to the solution's implementation, a structured evaluation of the obtained solution is performed under the same conditions and criteria defined for the measurement questions applied to the previous dataset. This evaluation encompasses a comparison between manual and automatic integration, allowing us to conclude whether the prototype's performance and usability deviates from the manual integration of flakiness detection and repair.

Table 1 - Measurement questions

MQ	Motivation	Metrics
MQ1. To what extent do the outcomes produced by detection tools align with the prerequisites of existing repair tools in addressing order-dependent flaky tests?	While approaches to detect and repair categories of flakiness exist, each has its unique input and output specifications. Currently, there is a lack of standardized formats for these tools, resulting in the necessity to investigate the compatibility between detection and repair approaches. This question intends to understand the compatibility between various tools better.	Compatibility score
MQ2. Can a unified approach for detecting and repairing flakiness caused by order-dependent tests be as effective as the existing alternative approaches?	The usefulness of a unified approach capable of detecting and repairing flaky tests depends on its capability of integrating the tools while maintaining the accuracy of the tool pair. This question aims to explore the effectiveness of the unified solution when compared to the effectiveness of the encapsulated tools while isolated across the same dataset.	Detection Recall Repair Recall
MQ3. Can a unified approach for detecting and repairing flakiness caused by order-dependent tests be less time-consuming than the existing alternative approaches?	A unified approach intends to maintain or decrease the current necessary amount of manual intervention from the developer to be capable of executing both tools. This question aims to explore the potential time-saving benefits achievable through the adoption of a unified strategy between the tool pairs.	Execution time
MQ4. Can a unified approach for detecting and repairing flakiness caused by order-dependent tests have better usability than existing alternative approaches?	For a unified approach to be effective in real-world applications, its usability should surpass that of the individual tools used in isolation. This question intends to evaluate the overall ease of use of the unified solution when compared to its isolated components, determining its viability for real projects.	Number of execution steps

1.4 Ethical Considerations

This dissertation respects IPP's code of ethics (Polytechnic Institute of Porto, 2020). That is to say that no plagiarism nor auto-plagiarism of any form has been performed. Any ideas, phrases, paragraphs, or complete texts from third parties, peers, or authors are cited and credited. All the work presented in this dissertation that has not been indicated otherwise was performed by the author. All obtained results are transparent and genuine, being publicly available (Mendes, 2024b). The available data includes all files on dataset selection, multi-criteria decision analysis, hypothesis testing, and the output resulting from all tool executions.

Additionally, as an author and student of a master's in informatics engineering, specialization area of software engineering I follow the joint ACM/IEEE code of ethics (Gotterbarn, Miller and Rogerson, 1997). Thus, all goals proposed in this dissertation are clearly defined, achievable, and within the author's capacity. Due to the experimental nature of the dissertation, the analysis, comparison, and subsequent usage of third-party projects and datasets is contemplated. As such, only projects and datasets with licenses explicitly permitting third-party use are considered for inclusion. Open-source projects or those accompanied by licenses granting express permission for external use are the focus of this study. It is relevant to note that most available tools allow their use but not modification, which was taken into consideration.

1.5 Structure

This document is structured as follows:

- Chapter "Introduction" contextualizes and exposes the problem, leading to the objectives of the study and identifying the research methodology.
- Chapter "Background" provides context for the subsequent research, describing the principles of maintainable test code, detailing what constitutes a flaky test, and analyzing the timeline of awareness of flakiness as a problem and its impacts on testing maintainability.
- Chapter "State of the Art" details the performed systematic literature review to answer four relevant research questions serving as a basis for the rest of the dissertation. The section provides an overview of the obtained results and synthetizes the obtained knowledge related to flakiness.
- Chapter "Empirical Evaluation of Approaches" details the selection of the testing dataset, empirical compatibility evaluation between the tools, and empirical isolated tool performance evaluation when detecting and repairing flakiness.
- Chapter "Conclusion" summarizes the findings obtained in the literature review and planned future work.
- "Appendix A" details the relevant measurement metrics.
- "Appendix B" details the SLR findings extensively.

- “Appendix C” details the dataset used in the controlled experiment.
- “Appendix D” details the definition of the compatibility evaluation criteria.
- “Appendix E” details the execution results of the evaluated tools.

2 Background

Software testing is a crucial aspect of the software development process, protecting against defects and ensuring the reliability and functionality of applications. In this chapter the importance of maintainable test code and its deriving principles are described, an overview of flakiness is given and its impacts on the maintainability and effectiveness of testing are explored.

2.1 Principles of Maintainable Test Code

Within the realm of software testing, the quality of test code is paramount for effective and efficient testing. This section delves into the fundamental principles that constitute maintainable test code. Understanding and adhering to these principles enhances the reliability of tests and contributes to the overall robustness of the software development process. According to (Aniche, 2022) test code should follow the following principles:

- **Tests should be fast:** Whenever maintenance or evolution is performed on the source code, the feedback of the tests is used to determine whether the system is working as expected. Slow test suites force developers to run the tests less often, making them less effective.

- **Tests should be cohesive, independent, and isolated:** Ideally, a single test method should test a single functionality or behavior of the system. Complex test code reduces our ability to understand what is being tested at a glance and makes future maintenance more difficult. Moreover, tests should not depend on other tests to succeed. The test result should be the same whether the test is executed in isolation or together with the rest of the test suite.

- **Tests should have a reason to exist:** Tests should either help find bugs or document behavior. If a test does not have a good reason to exist, it should not exist. All tests must be maintained therefore the perfect test suite is one that can detect all bugs with the minimum number of tests.

- **Tests should have strong assertions:** Tests exist to assert that the exercised code behaved as expected. Writing good assertions is therefore key to a good test. In cases

where observing the outcome of behavior is not easily achievable, the CUT should be refactored to increase its observability.

- **Tests should break if the behavior changes:** Tests let us know if the expected behavior breaks. If the behavior breaks and the test suite is still passing, something is wrong with the tests.

- **Tests should have a single and clear reason to fail:** Tests that fail indicate problems in the code, this failure is the first step towards finding the bug and the test code should help understand its cause.

- **Tests should be easy to write:** There should be no friction when it comes to writing tests. If a test is hard to write, it's easy for developers to give up and not do so. Ensuring developers have access to infrastructure that facilitates testing is paramount.

- **Tests should be easy to read:** Developers tend to spend more time reading code than writing it. Considering that tests should have a clear reason for failure; they should be easy to read to understand and identify the reason as quickly as possible.

- **Tests should be easy to change and evolve:** Production code changes and most of the time that forces the tests to change as well. For this reason, the test code should be implemented while ensuring that changing it will not be too difficult.

- **Tests should be repeatable and not flaky:** A repeatable test gives the same result no matter how many times it is executed. Tests that do not present this behavior are deemed flaky.

2.2 Overview of Flaky Tests

Flaky tests or non-deterministic tests are usually defined as tests that, executed without changing the test code and the CUT, variably pass and fail. The reasons leading to flaky behavior are varied and hard to pinpoint, making it hard to know whether a test is failing because the behavior is buggy or because it is flaky.

Flakiness, a prevalent issue in software testing, hampers numerous major companies, including Google (Micco, 2017), Microsoft (Lam, Muslu, *et al.*, 2020) and Apple (Kowalczyk *et al.*, 2020). It also affects smaller companies and open-source projects (Parry, Hilton, *et al.*, 2022; Parry, Kapfhammer, *et al.*, 2022). Although this is a widespread issue, research on the topic has only started growing recently.

One of the first publications referring to flakiness was (Fowler, 2011) which addresses the potential issues with non-deterministic tests. So far, three literature reviews focused on test flakiness have been made available (Parry *et al.*, 2021; Zolfaghari *et al.*, 2021; Tahir *et al.*, 2023). All of them detail a growing interest in flakiness starting in 2017 that keeps rising to date.

2.3 Impact of Flaky Tests

Flaky tests have a variety of negative consequences on testing. (Aniche, 2022) indicated the insurance that tests are repeatable and not flaky as a principle of maintainable test code, adding that flakiness leads to the loss of confidence in the test suites and said lack of confidence can encourage the deployment of systems whose tests fail. Flaky tests, however, can also lead to the violation of other important principles:

- **Tests should be fast:** Even if tests are fast the presence of flakiness may lead developers to rerun tests in case, they fail due to flakiness. This implicates added time required for testing since it forces repeated test suite executions.

- **Tests should be cohesive, independent, and isolated:** While tests may be intended to be completely independent and isolated from each other, occasional hard-to-detect shared states (a global variable for example) may remain in the test code. This leads to a category of flakiness called order-dependency. This is the most prevalent cause of flakiness. A detailed description of this category can be found in Section 3.2.2.1.

- **Tests should break if the behavior changes:** Given the nature of flaky tests to exhibit variable outcomes for the same code test and CUT, modifying the behavior of the CUT for a known flaky test could lead developers to believe the code is working as intended even if the test fails.

- **Tests should have a single and clear reason to fail:** Even if a test is correctly structured and intended to fail for a specific reason, flakiness may manifest with hard-to-pinpoint root causes.

Since flakiness has such a widespread impact on software testing, developers who often deal with flaky tests lose confidence in the testing process and, due to the effort required to identify and fix the problem, may sometimes ignore the failing tests. This outlines the importance of providing developers with tools capable of automatically detecting and repairing flakiness: diminishing the required time to deal with the issue, preserving confidence in testing, and ensuring that builds containing legitimate errors in the CUT are not deployed on the assumption those errors are merely instances of flakiness.

3 State of the art

This chapter describes a systematic literature review of the most current flaky test detection and repair tools and methodologies.

3.1 Methodology

In this section, a detailed description of the methodology used in the literature review is presented, detailing the research questions of interest to this study. To that end, the scope of the survey, the inclusion and exclusion criteria, and the screening process are described.

3.1.1 Research Questions

The first step was to define the research questions intended to build a foundation of knowledge to address the defined goals. These research questions guide a systematic literature review.

RQ1: What are the causes and associated factors of flaky tests? Answered in Section 3.2.2, this question explores flakiness and its root causes. The answer addresses various definitions of flakiness from multiple sources, exposes a deep understanding of order-dependency flakiness, and goes on to explore domain-specific causes of flaky tests. To better address the question, it was subdivided into the following sub-questions:

- RQ1.1: What are the causes of order-dependent tests?
- RQ1.2: What are the general causes of flaky tests?
- RQ1.3: What are the domain-specific causes of flaky tests?

RQ2: What approaches are available for detecting flaky tests? Answered in Section 3.2.3, this question concerns the currently available approaches to detect flakiness. The answer categorizes detection tools as dynamic or static and subsequently examines the available literature to identify approaches addressing the category of order-dependency. It then explores detection approaches concerning other categories of flakiness. To better address the question, it was subdivided into the following sub-questions:

- RQ2.1: What are the available approaches to detect order-dependent flaky tests?

- RQ2.2: What are the available approaches to detect other categories of flaky tests?

RQ3: What approaches are available for repairing flaky tests? Answered in Section 3.2.4, this question concerns the currently available approaches to repair flakiness. The answer analyzes the literature to identify approaches concerning the flakiness category of order-dependency and then further explores approaches oriented towards other categories. To better address the question, it was subdivided into the following sub-questions:

- RQ3.1: What are the available approaches to repair order-dependent flaky tests?
- RQ3.2: What are the available approaches to repair other categories of flaky tests?

RQ4: What are the benefits and drawbacks of a unified approach for detecting and repairing flakiness caused by order-dependent tests? Answered in Section 3.2.5, this question explores the feasibility and necessity of a unified solution for addressing order-dependency flakiness. The answer addresses the developer's need for an automatic approach to handle flakiness and identifies solutions currently available in the literature. To better address the question, it was subdivided into the following sub-questions for the SLR:

- RQ4.1: Is there a need for a unified solution?
- RQ4.2: Are there unified solutions currently available?

While these sub-questions allow for further insight into unified flakiness solutions, they only address RQ4 partially. Thus, to completely answer RQ4 further study guided by the outlined GQM process is necessary.

3.1.2 Research Scope

The data sources included in the literature review can be found in Table 2.

Table 2 - Selected Databases

Database	**URL**
ACM Digital Library	https://dl.acm.org/
B-On	https://www.b-on.pt/
Science Direct	https://www.sciencedirect.com

A research query to be performed on all selected databases was constructed based on relevant terms derived from the research questions. The query terms were intended to cover general test flakiness whether addressed as such or as non-determinism applied to the area of software development with the intent to either detect or repair it. The resulting full research query was *("flaky test" OR "test flakiness" OR "nondeterministic tests" or "non deterministic tests" OR "nondeterministic test" OR "non deterministic test") AND software AND (detect OR correct OR repair)*.

3.1.3 Eligibility Criteria

The eligibility criteria, found in Table 3, allowed me to narrow the relevant studies for our research and made the study selection process much easier. The inclusion criteria were based on the research questions for the literature review while the exclusion criteria excluded studies not relevant to the research. Due to the considerable amount of current research on the topic of test flakiness, the research was limited to academic studies released since 2020. However, studies released previously concerning relevant information were still included using the snowballing procedure (Wohlin, 2014). This facilitated an examination of the state of the art in the field while still including foundational studies.

The process followed the examination of the inclusion and exclusion criteria, if a study did not fulfill every single inclusion criterion or met any exclusion criterion it would not be included in the review (MacDonald, 2014).

Table 3 - Inclusion and Exclusion Criteria

Type	Criteria
Inclusion	Discusses the causes of flaky tests
	Discusses strategies or tools to detect or ease the detection of flaky tests
	Discusses strategies or tools to limit the negative impact or repair flaky tests
	Discusses strategies or tools that unify the detection and repair of flaky tests
Exclusion	Not a peer-reviewed publication
	Not published in English
	Not related to the field area of Software Engineering
	Published after January 2020

3.1.4 Selection Process

The selection process for relevant studies was performed as suggested in Figure 1. As such, the process consists of screening the title, moving on to the abstracts for the studies where relevance was not possible to judge from the title alone, and finally examining the full text of studies based on the relevance of the abstracts to the inclusion criteria. Reasons for the exclusion of the studies in the full-text screening are to be recorded (MacDonald, 2014).

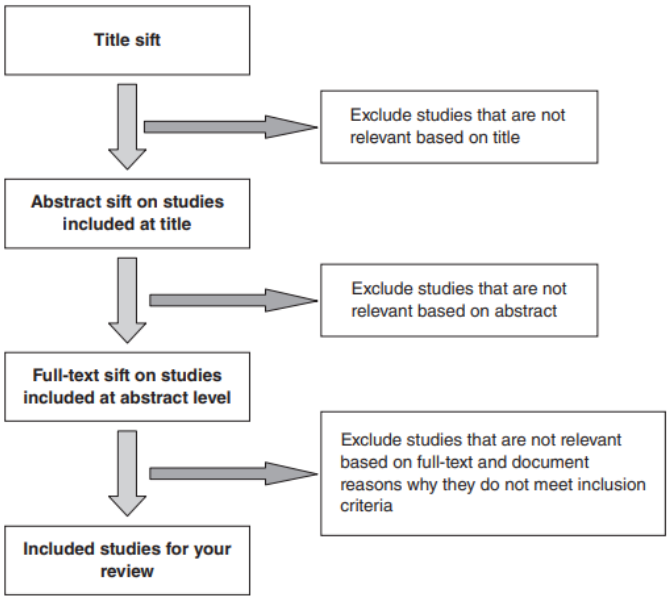


Figure 1 - Process of selecting studies

3.1.5 Data Collection Process Overview

Figure 2 shows a slightly altered PRISMA (Page *et al.*, 2021) diagram detailing the systematic literature review process. The Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) is a reporting guideline designed to improve the transparency and completeness in systematic reviews, helping authors provide clear and comprehensive information about the review process. Its steps are as follows:

- **Identification** consisted of obtaining all the studies from the data sources using the research query and applying the exclusion criteria. This yielded 212 valid studies, 18 of which were duplicated between databases.
- **Screening** was performed as detailed in the section 3.1.4 yielding 30 relevant studies.
- **Inclusion** consisted of thoroughly analyzing the selected studies to obtain the information relevant to the research.

The peer-review process was considered to constitute a sufficient quality bar, due to this no additional quality assessment stage was included in the performed methodology.

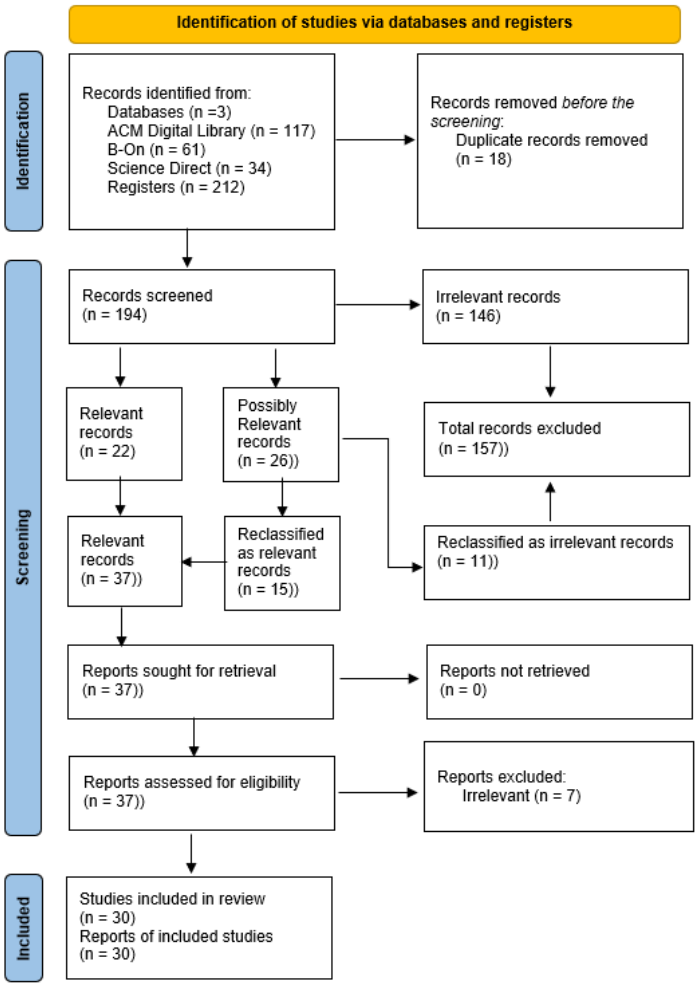


Figure 2 - Prisma diagram

A detailed list of the studies accepted in the full-text screening can be found in Table 31, a list of the excluded studies in Table 32 and a list of the studies subsequently added via snowballing in Table 33 (see Appendix B for details).

3.2 Results and Analysis

In this section, the results obtained from reviewing the relevant works are used to address the research questions. Furthermore, an overview of the findings is presented aiming to provide a better understanding of the focus of the most recently performed research.

3.2.1 Overview of the Publications

The research community has been paying rising attention to flaky tests in the past 6 years (Tahir *et al.*, 2023). This has led to the question of whether research trends have been maintained with those identified in previous literature reviews. The studies selected for this review span up to 2020 and are thus more recent than the timelines of previous reviews. An analysis of the accepted studies within this review aims to verify if the situation has been maintained.

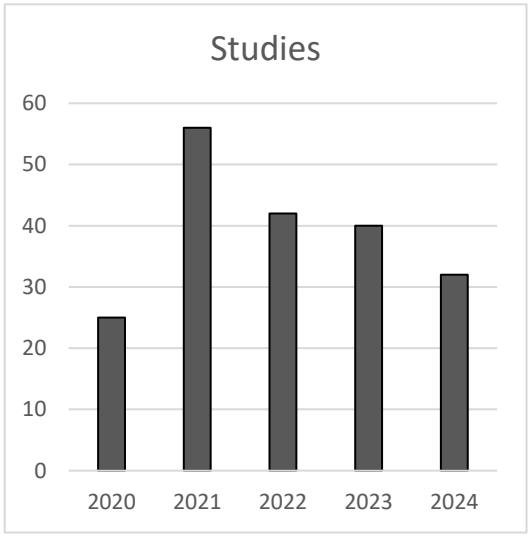


Figure 3 - Number of studies per publication year

Previous reviews have ascertained a growing interest in test flakiness, as indicated by the increasing amount of literature published each year (Parry *et al.*, 2021; Zolfaghari *et al.*, 2021; Tahir *et al.*, 2023). Meanwhile, analyzing the sources obtained in the Identification step of the data collection process (detailed in 3.1.5) and representing it in Figure 3, a peak in interest can be observed in 2021 with a decrease in 2022, 2023, and 2024. It should be noted that the referenced literature reviews were performed with a general flakiness research query rather than the one specified in this study, therefore while a peak of interest in 2021 can be observed when considering detection and repair methodologies for flakiness, this may not be the case when applied to general flakiness research. Additionally, the further decrease in 2024 can be explained by the year not being over at the time of this review.

Table 4 - Number of used languages in reviewed studies

Language	# Studies
Java	15
Python	6
.NET	1
Other	2
Multiple	4

In terms of programming languages, taking into account the studies approved in the Screening step of the data collection process (detailed in 3.1.5), it can be observed in Table 4 that Java was by far the most used language (accounting for 57% of the studies referring languages) whether for project evaluations or tool and methodology development. This aligns with the results of previous literature reviews (Tahir *et al.*, 2023) reinforcing Java as the leading language for flakiness research. Studies obtained via snowballing were not included in the count.

When considering the research questions leading to the inclusion of studies in this review, it can be observed that most studies focus on methods to detect flakiness (with 77% of included studies addressing RQ2), followed by identifying the root causes of flakiness, with a seemingly lower interest in facilitating the repair of those tests and finally the lowest amount of studies referring to the handling of flakiness fully automatically. The quantity of studies addressing each research question can be observed in Table 5. Literature reviews, albeit not introducing original tools or causes of their own were included in the count. Studies obtained via snowballing were not included in the count.

Table 5 - Number of studies addressing research questions

Research Question	# Studies
RQ1 - Causes	11
RQ2 - Detection	23
RQ3 - Repair	6
RQ4 - Unification	4

3.2.2 RQ1. What are the causes and associated factors of flaky tests?

In this section causes for flaky tests were analyzed from the selected academic studies. Different categories were identified, with some being grouped under the same category based on their characteristics.

3.2.2.1 What are the causes of order-dependent tests?

Order-dependent tests (OD) consist of tests that rely on shared values or resources that may be altered by preceding tests changing their outcome. As a result, when the run order of the tests is changed inconsistent outcomes are produced (Parry *et al.*, 2021). According to (Lam, Winter, *et al.*, 2020) order-dependent (OD) tests can be divided into 3 categories:

- Order-dependent brittle (OD Brit) - tests that fail when run in isolation but pass when run after some specific tests.
- Order-dependent victim (OD Vic) - tests that pass when run in isolation but fail when run after some specific tests.
- Non-deterministic (ND) - tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies.

In the case of OD Vic, tests that cause the change in the shared state and as a result, cause the test to fail, are called “polluters”.

(Wei *et al.*, 2022) expand on the definition, referring to tests that modify the shared state but may or may not have a victim in the current test suite as “latent polluters”. They then define non-idempotent-outcome (NIO) tests as a new category of OD tests. A test is an NIO test if the test outcome varies after repeated test runs, due to the changes in the shared state during the runs of the NIO test. This makes NIO tests both “latent-polluters” and “latent-victims”.

Table 6 - Order dependence shared state flakiness inducer operations (Chen *et al.*, 2023)

Class	**Description**
Instance Variable Dependency	The shared state is an instance variable
Static Variable Dependency	The shared state is a static variable
Third-Party Framework Dependency	The shared state is through a third-party framework
Cached Status Dependency	The shared state is cached data
Database State Dependency	The shared state has items in the database
File Permission Dependency	The shared state is a file handle
Resource Availability	The shared state is a specific resource

The shared state flakiness-inducing operations have been further categorized by (Chen *et al.*, 2023) as detailed in Table 6.

3.2.2.2 What are the general causes of flaky tests?

Resource leak is a category that occurs when the CUT is not properly managing shared resources, such as not releasing an acquired database connection (Tahir *et al.*, 2023). Some sources, such as (Chen *et al.*, 2023) consider it to be a sub-category of test order-dependency.

I/O operations are a category that occurs when a test incorrectly handles an IO operation, such as not accounting for test failure when the disk has no free space or becomes full during file writing (Parry *et al.*, 2021). Similarly to resource leaks, this category can be considered a sub-category of test order-dependency (Tahir *et al.*, 2023).

Async Wait is a category consisting of tests that make an asynchronous call but do not wait properly for the returned result, leading to test flakiness (Camara *et al.*, 2021; Chen *et al.*, 2023). This flakiness may not only be present in the test itself but also in the CUT (Lam, Winter, *et al.*, 2020).

Concurrency is a category consisting of tests that start several threads that interact non-deterministically, causing undesirable behavior (Camara *et al.*, 2021). It may be caused by, for example in UI, an attempt to manipulate data or resources before they are fully loaded on the page (Sousa, Bezerra and Machado, 2023).

Randomness is a category consisting of tests whose assertions rely on randomly generated values or information causing them to be unreliable if the test does not consider all possible random values that can be generated (Camara *et al.*, 2021; Parry *et al.*, 2021; Sousa, Bezerra and Machado, 2023; Tahir *et al.*, 2023).

Time is a category that occurs when a test relies on the local system's time zone possibly leading to discrepancies in precision and time zone (Camara *et al.*, 2021; Tahir *et al.*, 2023).

Network is a category consisting of tests relying on networking. Considering the unpredictability of the network tests may fail due to the network being unavailable, being slower than usual, or because network resources might not be available (Lampel *et al.*, 2021).

Floating point is a category consisting of tests that depend on floating point operations, which can lead to discrepancies resulting in nondeterministic behaviors even for simple calculations (Camara *et al.*, 2021; Parry *et al.*, 2021; Zolfaghari *et al.*, 2021).

Unordered collection is a category consisting of tests that assume iteration orders for unordered collection. Since these collections have no specific order, the assumed order may or may not correspond to the actual iteration order, possibly resulting in test flakiness. (Camara *et al.*, 2021; Parry *et al.*, 2021; Zolfaghari *et al.*, 2021)

Test timeout is a category that occurs when tests are specified with an upper limit on their execution time and the execution itself exceeds the specified time (Parry *et al.*, 2021). This occurs as the tests grow over time without reconfiguring the max runtime value (Zolfaghari *et al.*, 2021). Depending on the scope of the time limit the flakiness may be further categorized:

- Test Case Timeout – when the time limit is applied to a test case.
- Test Suite Timeout – when the time limit is applied to the test suite.

Platform dependency is a category that occurs when a test depends on some functionality of a specific operating system, library version, hardware vendor, etc... (Parry *et al.*, 2021) such as assuming specific platform properties to run (Chen *et al.*, 2023) or caused by an error in the dependency itself (Sousa, Bezerra and Machado, 2023).

3.2.2.3 What are the domain-specific causes of flaky tests?

Some categories of flakiness are only present in certain domains, further exacerbating the issue. This includes user interface (UI) categories, such as **DOM Selector Problems** referring to structures used to locate elements like XPath leading to problems with element identification and location, and **Animation Timing Issues** where tests depend on the execution of an animation to perform the test or directly test the animation itself and are, as a result, sensitive to timing differences in execution environment (Sousa, Bezerra and Machado, 2023).

Another identified domain with a specific flakiness category is machine learning (ML), due to **Algorithmic Non-determinism** which refers to the inherent non-determinism in the algorithm being used in the tests (Dutta *et al.*, 2020).

3.2.2.4 Summary

The sheer number of flakiness causes and the difficulty in identifying and distinguishing between them leads to a lack of consensus in the categorization of flakiness, it should be noted however that a basis of common flakiness categories seems to have developed around the work of (Luo *et al.*, 2014) with multiple studies either accepting some or all of the proposed categories as fact. Furthermore, the existence of domain-specific flakiness categories suggests a need for further research in flakiness targeted towards specific domains and the development of specialized tools capable of handling the resulting categories.

3.2.3 RQ2. What approaches are available for detecting flaky tests?

Several studies have explored strategies to identify flaky tests, this section focuses on recent methods used to identify, locate, or reproduce causes of flakiness.

There can be identified two distinct types of approaches to detect flakiness: either using dynamic techniques or static techniques.

Dynamic techniques are centered around test execution, needing one or more runs to identify flakiness. The concept is simple, if the outcome of a test is inconsistent across runs it is deemed flaky (Parry *et al.*, 2021).

While test rerunning exhibits a degree of effectiveness, it is computationally expensive. Due to this, new approaches explore ways to minimize the frequency of test runs by facilitating the manifestation of flakiness or narrowing down the selection of tests that need to be executed. Depending on the technique, it may either be capable of detecting a single category of flakiness or of identifying the test category among several.

Static techniques do not involve test execution and seem to mostly consist of classification-based machine learning techniques to predict whether a test is flaky.

Some approaches, however, combine both static and dynamic techniques resulting in hybrid methodologies.

3.2.3.1 What are the available approaches to detect order-dependent flaky tests?

Among the categories of flakiness, the one with the most available detection techniques are order-dependent tests. One such tool is iDFlakies (Lam *et al.*, 2019), which randomizes the test order execution in each run, classifying flaky tests into either order-dependent or non-order-dependent. Two recent studies improved upon iDFlakies:

- (Li and Shi, 2022) adapted the tool to be less costly during regression testing. Relying on the results of a regression testing selection (RTS) technique as input coupled with the tool's analysis, IncIDFlakies selects which tests have been affected by recent changes and runs iDFlakies on the resulting selection.
- (Wei *et al.*, 2022) modified iDFlakies to allow the execution of tests twice In the same VM. This technique allows pre-empting flakiness by detecting NIO tests before the shared state pollution can manifest flakiness.

Rather than randomly shuffling orders like iDFlakies, (Li *et al.*, 2023) expand upon order-dependency flakiness by exploring several ways to produce test orders. The study is based on the work of (Wei *et al.*, 2021) which states that given a set of N tests, N permutations of those tests can be produced, such that for all pairs of tests (t_1,t_2) there exists a permutation where t_1 is positioned right before t_2 (no other element is positioned in between the two) and there exists another permutation t_2 is positioned right before t_1. With this insight in mind, (Li *et al.*, 2023) presented three strategies to produce test order:

- Tuscan Intra-Class: which produces orders covering all possible test-class pairs and all test pairs within each test class.
- Tuscan Inter-Class: which produces orders covering all possible test pairs, including between classes.
- Target Pairs: these consider all test pairs, and since polluter/victim relationships only happen if a test shares a state with another (their criteria is if the test pairs have at least

one static field in common), produce orders that cover the most uncovered test pairs sharing a state.

After analysis, they concluded that the most cost-effective strategy was Tuscan Intra-Class.

Another tool capable of detecting order-dependency flakiness is DTDetector (Zhang *et al.*, 2014). This tool offers several algorithms to detect OD tests, introducing two new approaches:

- **Exhaustive Bounded Algorithm** with a defined bound generates and executes k permutations for each test, resulting in n^k permutations. Running a subset of the permutations reduces the time necessary to detect OD tests.
- **Dependence-aware bounded algorithm** starts by executing each test in isolation to determine which fields they access or write to. It then prunes the possible permutations of test executions removing those that do not have overlapping fields. It executes a bounded amount of the remaining permutations similarly to the Exhaustive Bounded Algorithm.

Several flaky tests fail due to an alteration in the shared state. Removing the polluter code in the offending tests can lead to a reduction of order-dependent flaky tests. The PolDet tool (Gyori *et al.*, 2015) identifies the shared state polluters by comparing the state before and after text execution. If the shared state changes after the test runs, the test is classified as a polluter. (Yi *et al.*, 2021) demonstrate this technique's ease of application by applying it to Java Pathfinder, a software verification tool designed for Java programs.

3.2.3.2 What are the available approaches to detect other categories of flaky tests?

This question concerns detection tools capable of detecting specific categories of flakiness besides order-dependency and tools capable of detecting several categories simultaneously.

Rerun

The common methodology is to simply run the tests multiple times and observe flaky results. This is traditionally referred to as a rerun-based methodology. One recent study in the academic literature by (Gruber and Fraser, 2023) details Flapy, a tool used to detect flakiness via rerun that is capable of mining Python projects to create flaky test datasets.

Category-specific

Specific detection tools for other flakiness categories have also emerged. For instance, FlakeScanner (Dong *et al.*, 2021) addresses concurrency-related flakiness in Android applications. The concurrent event-driven model of Android apps, in which only the main (UI) thread can access GUI and process user events can lead to the testing thread and background threads sending events to the UI thread simultaneously, resulting in race-conditions between them. To detect this, FlakeScanner ensures different event execution orders across runs facilitating the manifestation of flakiness.

NonDex (Shi *et al.*, 2016) addresses flakiness caused by assumptions of determinism on non-deterministic specifications, encompassing categories such as unordered Collection and randomness. To detect flakiness, it introduces varying levels of non-determinism into Java methods identified as non-deterministic, this allows flakiness to be manifested sooner thus diminishing the required number of reruns to detect flaky tests.

Additionally, (Dutta *et al.*, 2020) found that projects making use of probabilistic, or machine-learning frameworks tend to have a higher amount of flakiness relating to algorithmic non-determinism. To tackle this, they present FLASH, a tool that performs multiple reruns with different seeds until the assertion results converge. Besides classifying tests that fail at least once as flaky, it also calculates each assertion's probability of failing to determine if the test is flaky.

Noise Introducers

Rerun is computationally expensive, as such, some approaches reduce the time taken to detect flakiness by introducing noise to the test environment.

Shaker (Cordeiro *et al.*, 2022) adds stressing tasks that compete with the test execution for the use of resources (CPU or memory) to force the manifestation of flakiness. This strategy allows fewer runs to be executed to find flaky tests.

ShowFlakes (Parry, Hilton, *et al.*, 2022), similarly adds stressing tasks, however, it implements additional stressors such as network speed and compiler version manipulation to further force flakiness manifestation.

Machine Learning Algorithms

In terms of static approaches, they seem to mostly consist of classification-based machine learning techniques.

Methodologies relying on machine learning algorithms use a variation of predictors to train a model that statically evaluates test flakiness. The studies (Pinto *et al.*, 2020; Alshammari *et al.*, 2021; Camara *et al.*, 2021; Pontillo, Palomba and Ferrucci, 2021, 2022) evaluate flakiness predictors by training several classifiers such as Decision Trees (Freund, Yoav, 1999) and Support Vector Machine (Noble, 2006) and testing them upon datasets encompassing several projects. (Pinto *et al.*, 2020) used a vocabulary-based approach to train the model relying on textual metrics. (Camara *et al.*, 2021) showed the capability of using test smells as predictors. Meanwhile (Alshammari *et al.*, 2021; Pontillo, Palomba and Ferrucci, 2021, 2022) proved the efficacy of using a mix of production and test code metrics, code smells and test smells.

Detection tools can be applied to CI systems as shown by (Lampel *et al.*, 2021). The system consists of the collection of job telemetry for the various jobs being performed in the CI, in case of job failure a trained classification model evaluates whether the failure was due to a bug or is intermittent (i.e. a network failure). Job features like run time and CPU load were used to train the classifiers.

Specification Language

Static approaches are not limited to machine learning. A strategy to minimize errors due to non-determinism is defining properties that can warn of possible errors during runtime. (Mudduluru *et al.*, 2021) define a type qualifier system for Java capable of verifying if non-deterministic types are being used in a deterministic manner. This allows the minimization of flakiness categories like unordered collection by warning the developer of the problem.

3.2.3.3 Hybrid Methodologies

Some techniques combine dynamic and static methodologies.

FLAST (Verdecchia *et al.*, 2021) is intended to be deployed in a CI environment. It implements a k-nearest Neighbor classifier (Altman, 1992) to predict flakiness. This classifier is a lazy learner meaning that rather than building a model, it directly learns from new knowledge making it adaptable to specific CI environments. The authors claim that FLAST is intended to filter the flaky tests and have the remaining tests detected by approaches like rerunning with fewer resources.

CANNIER (Parry *et al.*, 2023) is an approach combining rerunning-based flaky test detection technique machine learning models. It uses classification models to classify tests as either flaky if above a certain threshold or not if below another threshold, however, if the probability lies between the defined thresholds CANNIER delegates the test to the rerunning-based technique.

FlakeRepro (Leesatapornwongsa, Ren and Nath, 2022) can identify concurrency-related bugs causing test flakiness. Rather than intending to detect which tests cause flakiness, this approach allows for the reproduction of the externally visible symptom of a failed execution, helping developers analyze and resolve the cause of the problem. The system's components perform the following steps:

1. *FlakeAnalyzer* examines which "critical memory accesses" affect the target error message. FlakeRepro will only explore interleavings involving critical accesses.
2. The Instrumenter component of FlakeFinder enhances the test by adding code or functionality that allows the system to monitor and control the interleavings of critical memory accesses.
3. The Explorer component of FlakeFinder systematically explores the interleavings of the critical accesses by executing the instrumented test binaries. The exploration space is pruned by feedback of the executions: When an interleaving leads to the target error message, FlakeFinder stops and produces the interleaving that a developer can later use to reproduce the failure. If no outcome leading to the target error message is found FlakeAnalyzer deems it unreproducible.
4. FlakeRepro, given the interleaving takes steps to reproduce it.

3.2.3.4 Methodologies For Addressing Flakiness in Gray Literature

Some widely adopted industry tools and frameworks for flakiness detection can be observed in the gray literature as detailed in Table 7. Notably, most of the existing tools rerun only the failed tests. This approach introduces situations where if a flaky test does not manifest flakiness in the first test run it will be erroneously marked as non-flaky.

Tools capable of rerunning tests that did not provide a way to mark tests as flaky were not considered.

Table 7 - Industry tools and frameworks for flakiness detection

Tool/Framework	Detection strategy	Description	Reference
TestNG	Rerun	Provides a simple interface for retrying tests allowing for the customization of the retry logic.	(Beust, 2004)
Maven Surefire	Rerun	Allows users to set failed tests to re-run up to N times, marking tests as “Flake” if they pass at least one time.	(Apache Software Foundation, 2011)
Jenkins Flaky Test Handler	Rerun	Jenkins plugin that adds support for the "rerunFailingTestsCount" option of the Maven surefire plug-in.	(Jost and Tintillier, 2023)
pytest-rerunfailures	Rerun	Pytest plugin that adds support for rerunning failed tests up to N times.	(Klearman, 2023)

3.2.3.5 Summary

Flakiness detection strategies can be divided into three categories: dynamic, static, and hybrid. Dynamic methods are based around rerunning tests at least once, the inherent cost in rerunning tests leads to several studies detailing avenues of run minimization or diminishing the amount of tests run. For static methods, most approaches use machine learning models to predict flakiness although the predictors vary. Hybrid methodologies on the other hand combine both, usually executing classification models to reduce the number of tests evaluated in the subsequent rerun. The flakiness category that has more detection tools capable of detecting it is order-dependency with several different strategies to either minimize the required test orders or to predict polluters. A list of the identified detection tools can be observed in Table 8.

Table 8 - Detection tools

Output type	Flakiness categories	Methodology	Language	Tool name	Study
Flakiness type	OD, Unspecified NOD	Rerun (Isolated containers; Vary orders)	Python	Flapy	(Gruber and Fraser, 2023)
Flakiness type, OD Victim, passing order, failing order	OD, Unspecified NOD	Rerun (Vary orders)	Java	iDFlakies	(Lam *et al.*, 2019)
Flakiness type, OD Victim, passing order, failing order	OD, Unspecified NOD	Rerun (Vary orders; RTS)	Java	IncIDFlakies	(Li and Shi, 2022)
NIO test	OD	Rerun (Vary orders)	Java/Python	-	(Wei *et al.*, 2022)
Flaky test	Concurrency, UI	Rerun (Vary event schedules)	Java	FlakeScanner	(Dong *et al.*, 2021)
Flaky test	Randomness, Unordered Collection	Rerun (vary implementation)	Java	NonDex	(Shi *et al.*, 2016)
Flaky assertion, Probability of assertion failure	Algorithmic non-determinism	Rerun (Vary random number seeds)	Python	FLASH	(Dutta *et al.*, 2020)
Flaky test	Concurrency, Async Wait	Rerun (Noise inducer)	Java/Python	Shaker	(Cordeiro *et al.*, 2022)
Flaky test	Concurrency, Network, Async Wait, OD	Rerun (Noise inducer)	Python	ShowFlakes	(Parry, Hilton, *et al.*, 2022)
Polluter test	OD	Static (Shared state validation)	Java	PolDet (JPF)	(Yi *et al.*, 2021)
Flaky test	OD, NOD	Machine learning + Rerun	Python	CANNIER	(Parry *et al.*, 2023)
Flaky test	Unspecified	Machine learning	Java	FLAST	(Verdecchia *et al.*, 2021)
Flaky test	Flaky interleaving	Static (Model checking) + Rerun	.NET	FlakeRepro	(Leesatapornwongsa, Ren and Nath, 2022)
Flaky test	OD	Rerun (Vary orders)	Java	DTDetector	(Zhang *et al.*, 2014)

3.2.4 RQ3. What approaches are available for repairing flaky tests?

A few recent studies describe repair solutions for flakiness categories. Some categories, however, had no flakiness repair methodologies identified in recent academic literature, reinforcing the necessity for further research on the issue.

3.2.4.1 What are the available approaches to repair order-dependent flaky tests?

As denoted in RQ2, the category of flakiness with the most studies centered around it seems to be order-dependency. When applied to repair methodologies this trend is maintained.

(Shi et al., 2019) proposed the iFixFlakies tool which takes an order-dependent test, a test order where the test passes, and a test order where the test fails as input. Establishing the concept of "helper" as tests already on the test suite whose logic resets or sets the state for order-dependent tests to pass. This tool produces a patch by identifying and extracting the relevant code from the helper and then applying it to either the start of the victim or the end of the polluter. This approach effectively repairs order-dependent tests, having the limitation however of requiring helpers to exist.

While iFixFlakies requires the pre-existence of helpers, the ODRepair tool builds upon it by identifying both the victim and polluter and subsequently determining the shared state causing the pollution (only determines heap-state pollution reachable from static fields in Java). With the pollution cause determined, the test generator Randoop (*Randoop: Automatic unit test generation for Java*, 2024) generates the to-be helper functions. With the presence of helper functions iFixFlakies can be run effectively, thus resolving order-dependent tests both with and without helpers.

3.2.4.2 What are the available approaches to repair other categories of flaky tests?

A study presenting a tool oriented toward a specific domain was identified. FLEX (Dutta, Shi and Misailovic, 2021), aims to minimize flakiness in machine learning projects by updating test assertion bounds. These defined bounds are often arbitrary, being based on developer intuition rather than systematically selected. Due to algorithm non-determinism an extreme value may be produced, violating the assertion bound and constituting flakiness. FLEX addresses this by statistically obtaining the most probable assertion bounds using Extreme Value Theory (EVT) (de Haan and Ferreira, 2006) and then updating the flaky assertion with the new values.

3.2.4.3 Summary

Research seems to primarily focus on flakiness detection, only a few repair strategies can be identified with each oriented towards specific categories of flakiness. The lack of a general tool to address flakiness is expected considering the variety of flakiness manifestations. Nonetheless,

the existence of methodologies that address order-dependency flakiness opens avenues of exploration for tools that combine detection and repair approaches.

Table 9 - Flakiness repair tools

Input type	Flakiness categories	Language	Tool name	Study
OD Victim, passing test order, failing test order	Order-dependency	Java	iFixFlakies	(Shi *et al.*, 2019)
OD Victim, OD Polluter	Order-dependency	Java	ODRepair	(Li *et al.*, 2022a)
Flaky test	Algorithmic Non-determinism	Python	FLEX	(Dutta, Shi and Misailovic, 2021)

3.2.5 RQ4: What are the benefits and drawbacks of a unified approach for detecting and repairing flakiness caused by order-dependent tests?

Multiple detection and repair approaches have been documented in academic literature. These solutions, however, are often isolated and require manual intervention for repairing when using a detection tool or for detecting when using a repair tool. A unified approach would streamline the detection and repair process, offering developers a tool applicable to their projects that can seamlessly and autonomously address flaky tests.

3.2.5.1 Is there a need for a unified solution?

Identifying and solving flakiness is not a simple matter, (Lam, Muslu, *et al.*, 2020) in a study of a dataset of Microsoft projects found that while bug reports due to non-flaky tests took on average 13 days to solve, it took 18 days to solve flaky-related tests. Besides the considerable lengthier time to repair, developers often have difficulty identifying the cause of flakiness which sometimes leads to the application of a “fix” that does not truly remove the flakiness inducer. The difficulty of identifying the source of flakiness or devising a fix can also lead to developers ignoring the issue or simply deleting the flaky test (Parry, Kapfhammer, *et al.*, 2022) especially for developers who frequently deal with flakiness. The negative impact flakiness has on developers and the overall testing process could be alleviated with tools capable of automatically detecting and repairing flaky tests.

3.2.5.2 Are there unified solutions currently available?

Table 10 details the only two unified solutions that have been identified in the literature. Using the detection tool iDFlakies and the repair tool iFixFlakies, (Wang, Chen and Lam, 2022) developed an approach named iPFlakies which encapsulates both tool's functionalities allowing for the detection and subsequent patching of test flakiness. This tool adapted both iDFlakies and iFixFlakies for Python projects resulting in a unified solution capable of detecting and repairing order-dependency flakiness in Python tests.

DexFix (Zhang et al., 2021) was also identified. It is an approach focusing on randomness and unordered collection categories. The tool makes use of NonDex to detect tests displaying flakiness and implements specific test repair strategies depending on the cause. The applied patches focus on flakiness derived by wrong assumptions by the developer, such as assuming HashMap/HashSet iteration is deterministic. In such a situation DexFix would transform the Hash object into a LinkedHash object which has precisely defined iteration orders. This approach offers a unified solution capable of detecting and repairing Java flaky tests relating to randomness and unordered collection.

Table 10 - Tools offering a unified solution (both detecting and repairing flakiness)

Flakiness categories	**Language**	**Tool name**	**Study**
Order-dependency	Python	iPFlakies	(Wang, Chen and Lam, 2022)
Randomness, Unordered Collection	Java	DexFix	(Zhang *et al.*, 2021)

3.2.5.3 Summary

Multiple studies on the impact of flakiness reveal that both manually detecting and repairing it is ineffective and discouraging for developers. This problem could be alleviated with the aid of fully automatic tools capable of detecting and repairing flakiness. Some such tools exist, such as iPFlakies and DexFix which are proof of concept for the combination of detection and repair approaches offering a unified solution capable of addressing the challenges associated with flakiness in testing. While order-dependency is one of the most studied and widespread causes of flakiness the only existing unified solution addressing it is iPFlakies which focuses on Python tests and, since it uses iFixFlakies, is limited to repairing OD Victims. While most of the research and approaches on the topic of flakiness are oriented to Java there seems to be a lack of unified solutions capable of handling order-dependency tests in the language.

3.2.6 Conclusion

Examination of the literature shows that interest in the field of flakiness remains high, with several studies being published annually. While the definitions of flakiness and its classifications vary considerably between publications most of the literature relies on the foundational work of (Luo *et al.*, 2014). Research emphasis is placed on the category of order-dependency, the most pervasive category, having several tools dedicated to detecting it and most repair tools geared towards repairing it.

Only a few integrations of tools used to detect and repair flakiness exist, most focus on other categories than order-dependency or target programming languages other than Java, despite it being the language most tools address. With the need for unified solutions applicable to real-world software projects, exploring the viability of integrating available OD detection and repair approaches in Java projects would facilitate the work of software developers and open future avenues for integration research.

4 Empirical Evaluation of Approaches

This chapter analyzes and evaluates the datasets identified during the literature review, as well as the detection and repair tools, to determine the viability of each tool for integration into a unified solution under similar testing conditions. The gathered information facilitates the selection of an appropriate dataset for the controlled experiment and enables an analysis of the overall compatibility between detection and repair tools, thereby addressing MQ1 (Table 1). Subsequently, this evaluation allows for comparing the tools in terms of performance, laying the foundation for the subsequent design of a unified solution.

4.1 Dataset Selection

The selection of an appropriate dataset is crucial to ensure that detection and repair tools provide valid results. This section outlines the methodology employed for the dataset selection using the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) (Hwang and Yoon, 1981) to rank the identified datasets based on the defined criteria.

4.1.1 Identified Datasets

Across the analyzed academic studies, multiple datasets were utilized to test detection or repair tools for distinct flakiness categories. Table 11 details them. Most of the identified datasets focus on Java projects.

Table 11 - Identified flaky test datasets

Dataset	Categories	Language	# flaky tests	# projects	Study
iDFlakies	Order-dependency / Other	Java	422	694	(Lam *et al.*, 2019)
FLASH	Algorithmic non-determinism	Python	11	20	(Dutta *et al.*, 2020)
NonDex	Unordered Collection/ Randomness	Java	21	8	(Shi *et al.*, 2016)
DexFix	Unordered Collection/ Randomness	Java	275	200	(Zhang *et al.*, 2021)
CANNIER	Multiple	Python	89,668	30	(Parry *et al.*, 2023)
iDoft (Java Gradle)	Multiple	Java	697	68	(Lam, 2020)
iDoft (Java Maven)	Multiple	Java	6,387	423	(Lam, 2020)
iDoft (Python)	Multiple	Python	1,615	343	(Lam, 2020)
FlakeFlagger	Multiple	Java	811	24	(Alshammari *et al.*, 2021)
FLAST	Order-dependency / Other	Python	1,383	13	(Verdecchia *et al.*, 2021)
Shaker	Concurrency	Java	75	11	(Cordeiro *et al.*, 2022)

4.1.2 Selection Approach

With the intent of identifying a suitable dataset for evaluating eligible tools, the TOPSIS method was employed. TOPSIS operates on the principle of deriving a compromise solution, which balances proximity to the positive ideal solution and distance from the negative one. This compromise solution is determined by choosing the alternative with the shortest geometric distance from the ideal solution and the farthest geometric distance from the negative ideal solution. Obtaining these compromise solutions involves several steps. Initially, a set of alternatives is compared, and their scores are normalized for each criterion. Subsequently, the geometric distance between each alternative and the ideal alternative, representing the best score in each criterion, is calculated. By comparing these distances, TOPSIS facilitates the identification of the optimal solution.

4.1.3 Criteria Definition

The effectiveness of dataset selection in TOPSIS relies on the establishment of clear and relevant evaluation criteria. The selection of the criteria used in this analysis was guided by the goal of identifying datasets that best represent the characteristics of real-world scenarios, adequately challenge the detection and repair tools under evaluation, and provide enough information for a subsequent evaluation of tool performance.

The defined criteria encompass various aspects essential for evaluating dataset quality and relevance of OD flaky testing tools:

1. Dataset Size (Weight: 30%): The dataset should contain enough tests to facilitate a meaningful and varied analysis. While there is no upper limit, datasets with excessive tests can be scaled down through random sampling to ensure a manageable analysis within the established timeframe.

2. Captured Flaky Test Characteristics and Project Metadata (Weight: 25%): The dataset should include comprehensive information regarding each flaky test's characteristics and corresponding project metadata. This information is crucial for understanding the context and characteristics of the test and can be meaningful to enrich the evaluation of the tools.

3. Order-Dependency Size (Weight: 25%): This criterion evaluates the number of flaky tests explicitly categorized as OD within the dataset. While the evaluation of tools in handling NOD tests is also valuable, the primary focus of this research lies in understanding and addressing OD issues. Thus, it is crucial to ensure a substantial representation of tests specified as OD within the dataset to accurately assess the performance of detection and repair tools regarding this type of flakiness.

4. Project Diversity (Weight: 20%): A diverse dataset comprising tests from a wide range of projects enhances the robustness of tool evaluation across different contexts. Including tests from various projects ensures a comprehensive assessment of tool accuracy and effectiveness.

By defining these criteria and assigning the appropriate weights, compromise solutions can be identified and ranked to obtain the one closest to the ideal solution. The combination of these criteria provides a comprehensive framework for dataset selection, attempting to attain the dataset that most adequately represents real-world testing scenarios.

4.1.4 Results

Table 11 lists several datasets identified in the literature review. However, given the research's emphasis on tools specific to the Java language, datasets targeting other programming languages were excluded. Additionally, although the NonDex dataset contains Java flaky tests,

it is not publicly available, preventing its inclusion in this analysis. A list of the datasets considered for the analysis can be found in Table 12.

Table 12 - Datasets used in TOPSIS analysis

Dataset	Categories	# flaky tests	# OD tests	# projects	Study
iDFlakies	Order-dependency / Other	422	211	694	(Lam *et al.*, 2019)
DexFix	Unordered Collection/ Randomness	275	Not specified	200	(Zhang *et al.*, 2021)
iDoft (Java Gradle)	Multiple	697	67	68	(Lam, 2020)
iDoft (Java Maven)	Multiple	6,387	1769	423	(Lam, 2020)
FlakeFlagger	Multiple	811	Not specified	24	(Alshammari *et al.*, 2021)
Shaker	Concurrency	75	Not specified	11	(Cordeiro *et al.*, 2022)

After the datasets were identified and considered for analysis, the TOPSIS method was employed to rank their suitability for further investigation. The ranked results of the TOPSIS analysis are presented in Table 13. The analysis highlights the most promising datasets based on the predefined evaluation criteria, a higher value of relative closeness to the ideal solution (C_i^*) corresponds to a higher proximity to the ideal solution.

Table 13 - TOPSIS for dataset alternatives ranked by relative closeness to the ideal solution

Dataset	C_i^*
iDoft (Java Maven)	0,86
iDFlakies	0,34
iDoft (Java Gradle)	0,15
FlakeFlagger	0,12
DexFix	0,11
Shaker	0

Analyzed the results, the 'iDoft (Java Maven)' dataset emerges as the most suitable option, exhibiting a high closeness to the ideal solution. Despite its suitability, the extensive size of this

dataset poses practical challenges within the constraints of this dissertation. Therefore, random sampling was used to address this issue while maintaining a representative sample.

This approach allows for the dataset scope to be managed effectively while preserving the diversity offered by the original dataset. Additionally, adding a condition for selected projects to exhibit a minimum set amount of OD tests ensures the primary focus on order-dependency is maintained while still allowing the analysis of occurring NOD tests. The derived dataset can be found in Table 34 of Appendix C.

4.2 Compatibility Evaluation

To assess the practicality of merging detection approaches with repair, the potential integration compatibility of the available tools was ascertained. Integration, in this context, refers to a seamless combination of detection and repair approaches to detect and repair OD flaky tests. If the tools are wholly incompatible, integration would only be possible after applying considerable efforts to standardize the outputs and inputs of the available approaches. This section focuses on determining the compatibility between the OD tools identified in Chapter 3.

This is accomplished by establishing specific criteria for compatibility assessment, employing the TOPSIS methodology, and estimating the current average compatibility rate. The insights gained from this analysis inform a conclusion addressing MQ1, thus providing insights into the practicality of merging detection and repair approaches.

4.2.1 Selection Approach

In the context of tool compatibility assessment, TOPSIS was deemed to be a fitting approach. It offers a relative perspective on the compatibility of different tool combinations and facilitates the evaluation of compatibility against an ideal, perfectly compatible solution. Unlike the previous dataset analysis, which relies on hard numerical data, this assessment relies on ordinal variables. Ordinal variables are a form of categorical variable where values possess a meaningful order or rank, yet the differences between values are not consistently quantifiable. They allow for ordering from lowest to highest, but the numerical disparities between categories lack uniformity. For example, a rating of poor and medium have a clear ranking, yet the disparity between them is not quantifiably higher or lower than from medium to high.

The challenge in applying TOPSIS to ordinal variables arises because TOPSIS typically works with quantitative data, where the distances between data points are meaningful and can be calculated using arithmetic operations. However, with ordinal variables, such distances are not easily quantifiable due to inconsistent differences between categories. To address this issue, techniques like Generalized Distance Measure (GDM) which can handle ratio, interval, and ordinal scale variables may be used. Due to this, a modification of TOPSIS named GDM-TOPSIS (Wachowicz, 2011), is used in this analysis, leveraging the application of GDM to correctly handle the defined assessment criteria.

4.2.2 Criteria Definition

Following the selection of methodology, the next step was to establish specific criteria with which each tool combination could be evaluated. These criteria encompass various aspects relevant to compatibility, including but not limited to required data formats and software dependencies.

To determine the value of each tool combination for each criterion, an examination and execution process was undertaken. While not subjected to thorough testing at this stage, this process involved analyzing the tools' characteristics to ascertain their alignment with the defined criteria.

The criteria and their corresponding definitions are detailed in Table 14. Each criterion was assigned an evaluation range from Poor (1) to Excellent (4), reflecting the degree to which a tool combination meets the specified criterion. For clarity and reference, separate tables outlining the definitions for each level of every criterion are provided in Appendix A (Table 35 to Table 39).

Table 14 - Compatibility Criteria

Criteria	Definition
Content Relevance	This criterion evaluates the relevance and usefulness of the information provided by the detection tool's output for the repair tool's input. It assesses whether the output contains the necessary information required for the repair process to be effective, such as identifying flaky tests or relevant test suite information.
File Structure Compatibility	This criterion evaluates whether the file formats used by the detection tool's output match the file formats expected by the repair tool's input. It considers aspects such as file extension, encoding, and folder structure.
Data Representation Consistency	This criterion examines whether the data representation used in the detection tool's output is consistent with the data representation expected by the repair tool's input. It assesses factors such as data types, data structures (e.g., arrays, dictionaries), and the organization of the data. This includes how the data is divided or segmented, such as the format of data fields or hierarchical organization.
Execution Environment Compatibility	This criterion evaluates the compatibility of the execution environments required by the detection and repair tools. It assesses whether the operating systems, hardware configurations, software configurations, and other environmental factors necessary for tool execution align with each other.
Dependency Version Compatibility	This criterion evaluates the compatibility of the dependencies required by the detection and repair tools in terms of version compatibility. It assesses whether the versions of system-level dependencies, global dependencies, or other external dependencies used by each tool align with each other, ensuring smooth integration and operation without conflicts or compatibility issues.

The criteria were also weighted to prioritize the feasibility of executing the tool in similar environments (Execution Environment Compatibility and Dependency Version Compatibility) and to ensure any essential data the repair tools required to generate and apply the fixes was made available by the detection tools outputs (Content Relevance). This resulted in the following weighting:

- Content Relevance – 40%.
- Execution Environment Compatibility – 20%.
- Dependency Version Compatibility – 20%.
- File Structure Compatibility – 10%.
- Data Representation Consistency – 10%.

4.2.3 Results

In the evaluation process, all available detection and repair tools tailored for Java and order-dependent (OD) tests were considered. However, some qualifying tools were excluded due to the following reasons: IncIDFlakies was excluded as a publicly available version could not be found; FlakeScanner is domain-specific and would be incompatible with the selected dataset; FLAST was excluded due to its status as proof of concept with only a replication package available, therefore requiring a substantial amount of time to convert into a usable tool.

The approaches examined in terms of compatibility for detection and repair can be found in Table 15 and Table 16, respectively.

Table 15 - Detection approaches used in compatibility analysis

Tool name	**Study**
iDFlakies	(Lam *et al.*, 2019)
NonDex	(Shi *et al.*, 2016)
Shaker	(Cordeiro *et al.*, 2022)
PolDet (JPF)	(Yi *et al.*, 2021)
DTDetector	(Zhang *et al.*, 2014)

Table 16 - Repair approaches used in compatibility analysis

Tool name	**Study**
iFixFlakies	(Shi *et al.*, 2019)
ODRepair	(Li *et al.*, 2022a)

The scores attributed to the criteria of each combination can be found in Table 17, discerned through analysis of documentation, practical execution of the tools, and analysis of required inputs and output.

Table 17 - Approaches Compatibility Criteria Scores

Integration	**Criteria Scores**				
	Content Relevance	File Structure Compatibility	Data Representation Consistency	Execution Environment Compatibility	Dependency Version Compatibility
IDFlakies + iFixFlakies	4	4	4	4	3
IDFlakies + ODRepair	3	3	3	4	4
NonDex + iFixFlakies	2	1	1	4	3
NonDex + ODRepair	3	2	1	3	4
Shaker + iFixFlakies	2	1	1	4	3
Shaker + ODRepair	3	1	2	4	3
PolDet (JPF) + iFixFlakies	1	1	1	3	4
PolDet (JPF) + ODRepair	1	1	2	3	4
DTDetector + iFixFlakies	3	2	2	3	4
DTDetector + ODRepair	3	3	3	3	4

The compatibility results obtained from the GDM-TOPSIS method are presented in Table 18. In theory, a tool combination with perfect compatibility across all criteria (a score of 4 in every criterion) would achieve a score of 1. While no such perfect score is observed in our results, it's worth noting that the combination "IDFlakies + iFixFlakies" stands out with the highest score by a significant margin. This outlier is attributed to both tools being developed by the same team, intending them to be individually applicable but easily combinable.

Conversely, at the opposite end of the spectrum, are the combinations involving PolDet (JPF), which exhibit low compatibility scores. This is primarily due to PolDet (JPF) not providing relevant information usable by the studied repair tools. PolDet (JPF) identifies OD polluters while both considered repair tools require either OD victims or OD brittles as input, resulting in a lack of overlap between the two.

Table 18 – GDM-TOPSIS combination alternatives ranked by relative closeness to the ideal solution

Tool Combinations	Positive Separation (d_j^+)	Negative Separation (d_j^-)	C_i*
IDFlakies + iFixFlakies	0.06	0.75	0.92
IDFlakies + ODRepair	0.15	0.64	0.81
DTDetector + ODRepair	0.24	0.53	0.68
DTDetector + iFixFlakies	0.32	0.45	0.58
Shaker + ODRepair	0.35	0.42	0.55
NonDex + ODRepair	0.38	0.41	0.52
NonDex + iFixFlakies	0.63	0.17	0.21
Shaker + iFixFlakies	0.63	0.17	0.21
PolDet (JPF) + ODRepair	0.74	0.10	0.12
PolDet (JPF) + iFixFlakies	0.80	0.07	0.08

4.2.4 Addressing MQ1

MQ1 (see section 1.3) concerns the extent to which the outcomes produced by detection tools align with the prerequisites of the repair tools for order-dependent flaky tests. While quantifying compatibility involves subjective judgments and uncertainties, by using the results obtained in the analysis an average compatibility score of 47% can be estimated (due to an average closeness of 0.47 to an ideal solution). This score, representing the closeness to an ideal solution, suggests room for improvement within the field of order-dependency testing.

The relatively low average compatibility score underscores the potential benefits of defining and standardizing input and output formats for detection and repair approaches. Specifically, the outlier of high compatibility between tools developed by the same team highlights that high compatibility is achievable when deliberate integration efforts are made.

Exploring the various tools revealed additional insights. While standardization facilitates integration, cases like PolDet (JPF) demonstrate compatibility may be hampered by differences in the subtypes of OD tests identified and repaired by the approaches. Even with standardization, incompatible subtypes would persist, remaining wholly incompatible. This emphasizes the importance of further research and categorization of OD subtypes.

The analysis of the compatibility of individual repair tools within different combinations revealed further information. When isolating only the combinations involving iFixFlakies, the average compatibility was found to be 40%. Conversely, ODRepair achieved an average compatibility of 54% within its respective combinations. Excluding the 'IDFlakies + iFixFlakies' combination led to a lower average compatibility of 27% for iFixFlakies.

This comparison suggests that ODRepair exhibits higher compatibility when integrated with tools developed independently, as opposed to those specifically designed for seamless integration. Further investigation would be required to precisely identify the underlying factors influencing this difference in overall compatibility.

In summary, faced with a short degree of tool compatibility, this area of research could benefit from standardization efforts in an attempt to make individual tools equivalent to tools designed to be compatible. Further compatibility studies on areas, such as OD subtypes, and understanding the factors driving compatibility differences among repair tools, are necessary.

4.3 Performance Evaluation

The tool selection process for developing a prototypical integration requires careful assessment of their capabilities. While ensuring data compatibility is essential for successful integration, it is not a sufficient condition to ensure the integration can detect and subsequently repair flaky tests. This section outlines the performance evaluation of the previously identified tools aiming to facilitate the identification of the most optimal tool combinations.

With this in mind, the applied selection approach is outlined, its corresponding performance criteria are defined, and the evaluation results are analyzed to determine the ranking of the tools in terms of performance.

4.3.1 Selection Approach

To establish a performance metric for the flakiness tools, the TOPSIS methodology was once again used. This methodology allows for the quantification of the tools' performances when applied to the selected dataset. However, during the evaluation, it was observed that only one of the two considered repair tools can be executed using the data given by the identified datasets. iFixFlakies requires the input of both passing and failing orders of execution, which, of all the available detection tools, only iDFlakies produces. Due to this, an isolated evaluation of iFixFlakies is not possible. As a result, the TOPSIS methodology is only applied to detection tools.

4.3.2 Criteria Definition

A performance evaluation can contemplate several different metrics. Due to flakiness' inherent complexity to detect, the tools may identify tests not present in the dataset as flaky, these tests aren't necessarily false positives nor true positives and the process of confirmation of the validity of those results is outside of this study's scope. As such, the defined criteria account only for the tests already identified in the selected dataset while not making assumptions regarding the unconfirmed tests.

The criteria for detection tools are as follows:

1. Recall OD tests (Weight: 50%): The dataset indicates several OD tests, any OD test present in the dataset and detected by the detection tool is deemed a True Positive while any OD test present in the dataset that goes undetected is deemed a False Negative.

2. Recall NOD tests (Weight: 20%): Similarly, any NOD test present in the dataset and detected by the detection tool is deemed a True Positive, while any NOD test present in the dataset that goes undetected is deemed a False Negative.

3. Time taken (Weight: 30%): The duration of execution for the tools is also relevant considering some can take more than one day for one single project. This criterion consists of the time taken for the tool to execute all its input data from script start to end.

While TOPSIS is not applied to the analysis of repair tools, the criteria of analysis were still defined and collected:

1. Recall OD tests: The dataset indicates several OD tests, any OD test present in the dataset and detected by the detection tool is deemed a True Positive while any OD test present in the dataset that goes undetected is deemed a False Negative. Unlike with the detection tools, NOD tests are not considered due to the available repair tools only being capable of repairing OD Victims and OD Brittles.

2. Time taken: The duration of execution for the tools is also relevant considering some can take more than one day for one single project. This criterion consists of the time taken for the tool to execute all its input data from script start to end.

4.3.3 Results

The evaluation was performed under similar conditions for each tool, being executed in the same Linux environment on the same machine. The tools considered for evaluation were the same as in 4.2 except for PolDet (JPF) which was excluded due to its incompatibility with either of the available repair tools. Additionally, while DTDetector was verified to execute correctly

for its author's example, its behavior could not be reproduced for the other projects, producing no output. The tool was therefore also excluded from the analysis.

A summary of the obtained execution results for the detection tools can be found in Table 19. Tables detailing the results for each dataset project can be found in Appendix E.

Table 19 - Detection Tools Execution Results

Detection Tool	Detected OD tests	Detected NOD tests	Other detected tests	Total Recall (OD)	Total Recall (NOD)	Total Recall (All tests)	Time (mins)
IDFlakies	222	90	48	0,77	0,57	0,70	2495
NonDex	15	34	1700	0,05	0,22	0,11	1454
Shaker	2	20	8	0,01	0,13	0,05	1819

The collected results indicate that iDFlakies had the best recall for the OD and NOD tests present in the dataset while also taking the longest time to execute. The difference of detected dataset flaky tests and corresponding recall between IDFlakies and the two other tools is considerable, however the total execution time considerably exceeds the different tools. It should be noted that the "Apache Hive" project, present in the dataset, was interrupted after 24 hours of execution for Shaker, therefore a longer execution time may have detected more flaky tests.

Regarding repair, the only tool accepting the dataset data as input is ODRepair. The results of its execution can be found in Table 20, these present an overall recall of 0.16 for the analyzed dataset. While this recall is lower than the overall results found by the researchers responsible for ODRepair (Li et al., 2022a) indicate, the overlapping projects in the used dataset (namely "fastjson", "remoting" and "visualee"), are in line with their results. It is possible that a more comprehensive dataset would improve the recall to be in line with the original study.

Table 20 - ODRepair Dataset Execution Results

Project	OD tests in dataset	Generated Patch	Recall (OD)	Time (mins)
spring-cloud-netflix	10	0	0	9
json-iterator	20	0	0	8
nifi	21	0	0	143
fastjson	15	1	0,07	70
wasp	28	0	0	12
hive	19	0	0	74
spring-cloud-kubernetes	30	0	0	34
visualee	47	46	0,98	222
ormlite-core	87	0	0	40
remoting	10	0	0	26
Total Count /Mean Recall	287	47	0.1	638

Utilizing the detection tools' execution results, the TOPSIS method was employed to rank their suitability in terms of performance. The ranked results of the TOPSIS analysis are presented in Table 21. The analysis highlights the most promising detection tools based on the defined criteria.

Table 21 - TOPSIS for detection tool performance alternatives ranked by relative closeness to the ideal solution

Detection Tool	**C_i***
IDFlakies	0,85
NonDex	0,17
Shaker	0,10

The results confirm IDFlakies as the detection tool most suitable in terms of performance, exhibiting a considerably higher closeness to the ideal solution than the other alternatives.

4.4 Conclusion

By analyzing and systematically comparing the available datasets, the most suitable one, 'iDoft (Java Maven)', was selected as the sample basis for the controlled experiment. This enabled each tool to be executed using similar inputs during the performance evaluation. However, due to the inability to run any repair tool other than ODRepair in isolation, the performance evaluation was only applied to the detection tools. In terms of performance, iDFlakies proved to be the most effective detection tool in handling both OD and NOD flaky tests, achieving significantly higher recall rates at the expense of increased time compared to the alternatives.

Additionally, an evaluation of the overall compatibility between the available tools revealed that they exhibit an average compatibility of only 47% when compared to a hypothetical ideal solution. This compatibility score increases to 54% when considering the repair tool developed without specific integration in mind (ODRepair) and decreases to 40% for the repair tool designed to ease integration with a specific detection tool (iFixFlakies). This suggests that tools developed without a specific integration focus are easier to integrate with other, previously unforeseen tools.

The results of these analyses provided compatibility and performance scores that serve as a basis for deciding upon the ideal combination for developing a prototype.

5 Integration Prototype

In this chapter, the composition and design of the prototype are delineated. The results of its execution are analyzed and compared, providing a basis for addressing the measurement questions. In turn, the measurement questions are methodically answered using statistical analysis.

5.1 Integration Components Selection

Based on the results presented in Chapter 4, a TOPSIS analysis considering compatibility and performance can be applied to determine the preferable components to constitute a viable prototype.

Given that only the ODRepair repair tool was executable with the selected dataset, it is automatically chosen as the repair component of the prototype. The selection of the detection tool, however, depends on the TOPSIS analysis results, which assign equal importance to compatibility and performance. The alternatives taken into consideration can be observed in Table 22.

Table 22 - Identified alternatives for prototype selection

Integration	Compatibility Value	Detection Tool Performance Value
IDFlakies + ODRepair	0,81	0,85
NonDex + ODRepair	0,55	0,17
Shaker + ODRepair	0,52	0,10

The results of the analysis applied to these alternatives are presented in Table 23. It can be observed that IDFlakies is the preferred detection tool to combine with ODRepair. This selection represents the optimal solution in terms of both compatibility and performance. As such, these two tools will be utilized as the basis of the prototype.

Table 23 - TOPSIS for prototype alternatives ranked by relative closeness to the ideal solution

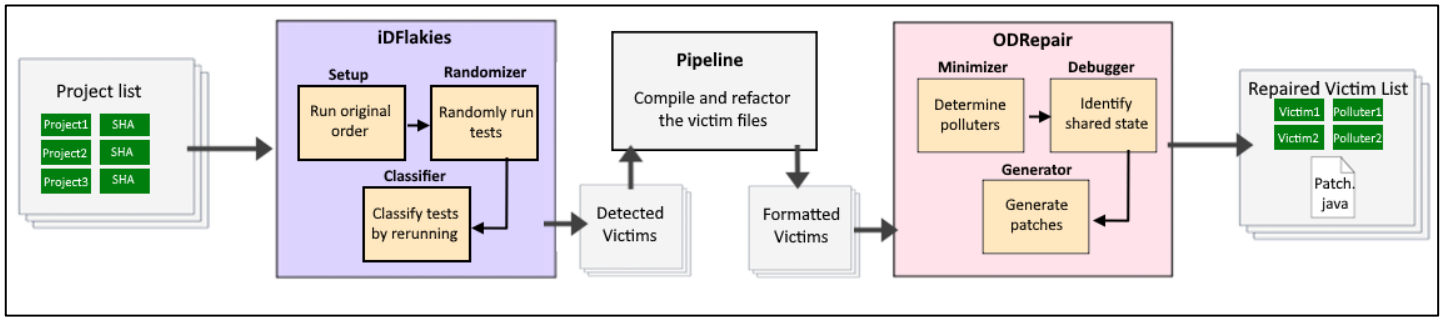
Integration	C_i*
IDFlakies + ODRepair	1
NonDex + ODRepair	0,09
Shaker + ODRepair	0

5.2 Design

Considering the determined prototype components, I propose intIDRepair (Mendes, 2024b) a simple integration between the detection and the repair tools aiming to simplify the process of detecting and repairing the flakiness of a given project while removing the necessity of user intervention during the process.

The integration is intentionally constructed to maintain the integrity of the component tools, making no internal changes in their structure, and providing a pipeline to bridge the component tools. As a result, intIDRepair encapsulates the functionalities of iDFlakies and ODRepair as shown in Figure 4.

Figure 4 - Overview of intIDRepair



5.2.1 iDFlakies

The integration utilizes iDFlakies' framework which, given a set of projects, isolates each one in a docker container and performs the following steps to detect flakiness:

1. Setup – The tool checks whether all tests of a module pass, if not, no further exploration is done for that module.
2. Randomizer – The tool executes the project's tests according to various randomized orders based on a user-specified number of rounds.
3. Classification – The tool reruns failing and passing orders for each test to categorize it as OD or NOD.

Once the detection process is complete, the produced output is organized into separate modules mirroring the analyzed project. The output lists the flaky tests, their category, and their respective passing and failing orders.

5.2.2 Pipeline

The pipeline unique to intIDRepair handles the processing of the output data produced by iDFlakies to format it into a valid input for ODRepair. To do this, it performs the following steps:

1. Gather the produced lists of flaky tests spread across various modules compiling them into a single file.
2. Refactor the qualified test names into valid formats while removing invalid or duplicate lines.
3. Append the tests' corresponding project URL and git SHA.

5.2.3 ODRepair

To repair the identified flaky tests, the integration executes ODRepair, which for each OD test identifies its polluter and attempts to generate a patch. To do so it performs the following steps:

1. Minimizer – Given the identified flaky test, ODRepair utilizes the Minimizer component from iFixFlakies to determine its polluter.
2. Debugger – Identifies the shared state that the polluter modifies in such a way that the order-dependent test fails in the failing test order.
3. Generator – With the OD test and its polluted static field identified in the Debugger, a patch can be generated using Randoop. When the patch is applied right before the OD test, it can pass when run after the polluter in the failing test order.

5.3 Results

With the prototype implemented according to the defined design, a performance evaluation was performed utilizing the dataset determined in Chapter 4. The results of its execution relating to OD tests can be found in Table 24. A more complete overview of the obtained results of all tests can be found in Table 43 of Appendix E.

Table 24 - intIDRepair Dataset Execution Results (OD tests)

Project	OD tests in dataset	Detected tests (OD)	Detection Recall (OD)	Generated Patches (OD)	Repair Recall (OD)	Time taken (mins)
spring-cloud-netflix	10	8	0,8	0	0	39
json-iterator	20	20	1	0	0	17
nifi	21	1	0,05	0	0	174
fastjson	15	0	0	0	0	112
wasp	28	28	1	0	0	226
hive	19	2	0,11	0	0	792
spring-cloud-kubernetes	30	29	0,97	0	0	456
visualee	47	47	1	45	0,96	227
ormlite-core	87	87	1	0	0	76
remoting	10	0	0	0	0	558
Total Count /Mean Recall	287	222	0.59	45	0.10	2,677

The total number of tests detected by iDFlakies (222 as found in Table 24) was equal to the number detected during its independent execution (222 as found in Table 19). This consistency is expected, given that the integration involved no modifications to iDFlakies. Meanwhile, ODRepair generated two fewer patches when using the pipeline data in the repair component (47 as found in Table 20 to 45 as found in Table 24).

In terms of time taken, the combined execution time of the isolated runs of iDFlakies and ODRepair ($Total\ iDFlakies\ execution\ time + Total\ ODRepair\ execution\ time = 3{,}784$ minutes) is greater than the execution time of the integrated run (2,677 minutes as found in Table 24).

5.3.1 Addressing MQ2

Regarding the matter of whether the unified approach can match the effectiveness of existing alternative approaches in detecting and repairing OD test flakiness, the investigation was divided into two components: detection and repair. The statistical significance of the previously observed decrease in recall in the unified approach was assessed through hypothesis testing and applying relevant statistical analyses. The null and alternative hypotheses defined for this purpose are presented in Table 25.

Table 25 - MQ2 Hypothesis Testing

Component	Type	Hypothesis
Detection	Null Hypothesis (H_0)	There is no difference in the effectiveness between the unified approach and existing alternative approaches for detecting order-dependent test flakiness.
	Alternative Hypothesis (H_1)	There is a difference in the effectiveness between the unified approach and existing alternative approaches for detecting order-dependent test flakiness.
Repair	Null Hypothesis (H_0)	There is no difference in the effectiveness between the unified approach and existing alternative approaches for repairing order-dependent test flakiness.
	Alternative Hypothesis (H_1)	There is a difference in the effectiveness between the unified approach and existing alternative approaches for repairing order-dependent test flakiness.

To evaluate either hypothesis, the Shapiro-Wilk test (Shapiro and Wilk, 1965) was used to assess the normality of the data, and the Wilcoxon Signed-Rank test (Wilcoxon, 1945) was employed to compare paired samples. The test results can be found in Table 26.

Table 26 - Performance Hypothesis Testing Results

Component	Test	Data	p-value
Detection	Shapiro–Wilk	Isolated detection recall	1×10^{-3}
		Integrated detection recall	1.4×10^{-3}
	Wilcoxon Signed-Rank	Isolated detection recall Integrated detection recall	5.9×10^{-1}
Repair	Shapiro–Wilk	Isolated repair recall	2.4×10^{-7}
		Integrated repair recall	1×10^{-7}
	Wilcoxon Signed-Rank	Isolated repair recall Integrated repair recall	1.8×10^{-1}

For the defined significance level of 5%, the results indicate no significant evidence of a difference in recall between the unified approach and the isolated components. Consequently, the null hypotheses are accepted, leading to the conclusion that the integration of the detection and repair components neither diminished nor enhanced the approach's effectiveness. All hypothesis testing data and results can be found in (Mendes, 2024b).

5.3.2 Addressing MQ3

A similar methodology to MQ2 was employed to determine whether the unified approach is less time-consuming than existing alternative approaches. To evaluate the impact of manual integration on the time required in the non-integrated approach, a manual estimation of the time needed to transform the detection tool's output data to a usable format for the repair tool was performed. The results of these estimations are presented in Table 27. It is important to note that these estimations were conducted by the author, who had prior knowledge of the required steps. Consequently, the time needed for a developer without prior knowledge could be considerably higher.

Table 27 - Manual Integration Time

Project	Manual Integration Time (mins)
spring-cloud-netflix	7
json-iterator	11
nifi	No detected OD tests
fastjson	No detected OD tests
wasp	14
hive	3
spring-cloud-kubernetes	17
visualee	7
ormlite-core	12
remoting	No detected OD tests

The statistical significance of the observed reduction in time was assessed through hypothesis testing. The defined hypotheses are detailed in Table 28.

Table 28 - MQ3 Hypothesis Testing

Condition	Type	Hypothesis
Includes the manual integration time. Only contemplates projects where OD tests were detected.	Null Hypothesis (H_0)	There is no difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.
	Alternative Hypothesis (H_1)	There is a difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.
Does not include the manual integration time. Only contemplates projects where OD tests were detected.	Null Hypothesis (H_0)	There is no difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.
	Alternative Hypothesis (H_1)	There is a difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.
Does not include the manual integration time. Contemplates all projects.	Null Hypothesis (H_0)	There is no difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.
	Alternative Hypothesis (H_1)	There is a difference in the time taken between the unified approach and existing alternative approaches for detecting and repairing order-dependent test flakiness.

To evaluate the hypotheses, the Shapiro-Wilk test was used to assess the normality of the data. When normality was verified, Student's paired t-test (Student, 1908) was applied to compare the paired samples. The test results can be found in Table 29.

Table 29 - Time Taken Hypothesis Testing Results

Condition	Test	Data	p-value
Includes the manual integration time. Only contemplates projects where OD tests were detected.	Shapiro–Wilk	Sum of isolated execution times plus manual integration time	1.3×10^{-1}
		Integrated execution time	1.3×10^{-1}
	Paired t-test	Sum of isolated execution times plus manual integration time Integrated execution time	1.7×10^{-2}
Does not include the manual integration time. Only contemplates projects where OD tests were detected.	Shapiro–Wilk	Sum of isolated execution times	1.2×10^{-1}
		Integrated execution time	1.3×10^{-1}
	Paired t-test	Sum of isolated execution times Integrated execution time	6.2×10^{-2}
Does not include the manual integration time. Contemplates all projects.	Shapiro–Wilk	Sum of isolated execution times	2.3×10^{-1}
		Integrated execution time	1.1×10^{-1}
	Paired t-test	Sum of isolated execution times Integrated execution time	3.9×10^{-1}

For the defined significance level of 5%, the results indicate no significant evidence of a difference in recall between the unified approach and the isolated components when the manual integration time is not contemplated. This applies whether all projects or only the ones where OD tests were detected are included. However, if the manual integration time is included a statistically significant difference is identified.

Consequently, we fail to reject the null hypothesis if the manual integration time is not included. This indicates that executing both components in an isolated matter takes as much time as an integrated execution. However, the null hypothesis is rejected when the estimated manual integration times are included. This indicates that executing both components and accounting for the time taken to manually integrate them takes a statistically significant higher amount of time than using the integrated approach. All hypothesis testing data and results can be found in (Mendes, 2024b).

5.3.3 Addressing MQ4

To determine if a unified approach can have better usability than existing alternative approaches, the number of steps required to execute the detection component, transform its output into usable input by the repair component, and execute the repair component were compared to the number of steps required to run the unified approach.

To determine the number of steps required by both approaches, the steps required to previously evaluate the tools and to determine the manual integration time were utilized as a guideline. This analysis does not contemplate the installation requirements for each approach since the unified solution requires the same prerequisites as its components.

The steps to execute and integrate the tools separately are as follows:

1. Prepare an input CSV file with the intended projects.
2. Execute iDFlakies via the terminal using the projects file as input.
3. Once iDFlakies ends its execution, in each produced output file (one for each project) containing the identified flaky tests, append the project URL and SHA to each entry.
4. Compile all produced CSV output files containing the identified flaky tests into a single CSV file.
5. Refactor the qualified test names into valid formats.
6. Remove all invalid and duplicate entries.
7. Execute ODRepair using the previous CSV as input.

This results in 7 separate steps with an intervention required after the execution of iDFlakies, which could take several hours. Comparatively, the steps to execute the integration are as follows:

1. Prepare an input CSV file with the intended projects.
2. Execute intIDRepair via the terminal using the projects file as input.

This results in 2 separate steps and no interventions are required after a considerable amount of time passes. Besides the reduction in steps, the integration also produces as output both a file containing all detected flaky tests and a file containing all repaired flaky tests. Additionally, all arguments available to the component tools that change their execution parameters are also available in the integration.

As such, the integration displays a higher degree of usability when compared to the isolated execution of its component tools, providing the same flexibility in their execution while diminishing the interventions required by the user and facilitating the identification of which flaky tests were identified and repaired.

6 Conclusion

This chapter details the results obtained in the performed literature review related to all the research dimensions and the results obtained during the controlled experiment. Conclusions concerning the outlined goal and their respective contributions are presented and future work is discussed.

6.1 Results

This section addresses the results obtained for the research questions.

6.1.1 RQ1. What are the causes and associated factors of flaky tests?

The categories of flakiness are numerous, with varying definitions across sources. However, the study by (Luo et al., 2014) is the foundational reference for flakiness classification. The most prevalent flakiness category among the analyzed academic studies is order-dependency, which can be divided into sub-categories such as Brittles, Victims, or NIOs. Besides the commonly established flakiness categories, some are domain-specific, such as DOM Selector Problems for UIs and Algorithmic Non-determinism for machine learning projects.

6.1.2 RQ2. What approaches are available for detecting flaky tests?

Several approaches for detecting flakiness in tests have been documented in the literature. These approaches can be categorized into three main types: static, dynamic, and hybrid. Static approaches identify flaky tests without requiring test execution, dynamic approaches detect flakiness by executing tests multiple times, and hybrid approaches incorporate elements of both static and dynamic methods. Most of the detection tools developed so far are oriented towards Java projects, with Python being the second most targeted language. Given that order-dependency is the most common type of flakiness, many detection tools are specifically

designed to identify OD tests. Despite the advancements in these approaches, the tools remain largely experimental and require further research before they can be adopted for widespread use.

6.1.3 RQ3. What approaches are available for repairing flaky tests?

Automatic repair approaches for flaky tests are significantly less common than detection approaches, with only three distinct repair tools identified in the literature. Two of these tools focus on OD tests, reinforcing the trend that order-dependency is the most addressed flakiness category. These OD repair tools can fix both OD Victims and OD Brittles, provided the polluter can be identified. The third repair tool targets the repair of algorithmic non-determinism.

6.1.4 RQ4: What are the benefits and drawbacks of a unified approach for detecting and repairing flakiness caused by order-dependent tests?

The literature highlights a need among developers for a unified approach to automatically detect and repair flaky tests, as manual methods are highly unreliable. In response to this need, two unified approaches have emerged: iPFlakies, which targets order-dependent (OD) tests in Python projects, and DexFix, which addresses issues related to randomness and unordered collections in Java projects.

Concerning the potential benefits and drawbacks of unified approaches, the experiment showed that unifying the available detection and repair tools can maintain similar levels of recall to their isolated execution while possibly improving execution time. Consequently, such unifications can enhance operability, minimizing user intervention while maintaining performance levels. This indicates that tools capable of automatically detecting and subsequently repairing OD flakiness can be effectively created using existing tools as components.

However, the analysis also revealed that the overall compatibility of separate tools is low. This is especially true for repair tools designed to integrate seamlessly with specific detection tools. These tools show high compatibility with their intended counterparts but exhibit a marked decrease in compatibility with other tools compared to repair tools designed independently of specific detection approaches.

6.2 Contributions

The controlled experiment and associated analysis highlight the low overall compatibility between existing tools, emphasizing the need for standardization of inputs and outputs. The developed prototypical tool serves as a proof of concept, demonstrating that the usability of detecting and repairing flaky tests can be improved by integrating existing tools while

maintaining their corresponding recall and execution time. The resulting tool, along with all research data and tool execution results, are publicly available (Mendes, 2024b).

Additionally, during this study, the use of the tools led to the identification and submission of a pull request (Mendes, 2024a) that fixed a bug impeding the execution of iDFlakies. Another bug of iDFlakies, with a currently open issue (rRajivramachandran, 2023), was replicated when executing the Apache Hive project (Apache Software Foundation, 2023).

6.3 Threats To Validity

The results of this study regarding the detection and repair capabilities of the analyzed tools may not be generalizable to other projects. To address this limitation, a diverse dataset was selected using randomized sampling from a large publicly available dataset.

The evaluation of compatibility among the available OD tools relies on metrics that are inherently subjective and not easily quantifiable.

Due to a time constraint of 24 hours, some tools were unable to fully analyze all considered projects. Increasing the budget allocation could potentially alter the results obtained in this analysis.

If a repair tool indicated that flakiness was fixed, it was assumed to be accurate. The patches generated to address flakiness were not manually verified for effectiveness.

Furthermore, intIDRepair itself or its component tools may contain bugs that could influence the results. To mitigate this potential issue, the author conducted a thorough review of the code, execution logs, and input/output files.

6.4 Future Work

The results obtained from the performed empirical evaluations open avenues of research capable of facilitating the development of further integrations:

- The identifiably low compatibility between the available OD detection and repair tools demonstrates the lack of standardized patterns for the output/input formats of the tools. Such a standardization, if possible, would facilitate the future integration of newly available tools.
- In terms of the data in publicly available flaky test datasets, information vital for some tools' executions is missing, such as the case of iFixFlakies requiring the passing and failing orders of OD tests. A study of the input requirements of detection and repair tools could indicate what information each category of flakiness requires to enrich the available datasets.

In terms of the developed integration, further improvement possibilities can be explored. Studying and filtering tests that would likely not be repaired by the repair tool, such as excluding NOD tests from the input of ODRepair, could improve its execution time. Additionally, a further combination of a larger number of approaches, either detection or repair could be a possible way to enhance their overall recall.

References

Ahlgren, J. *et al.* (2021) 'Testing web enabled simulation at scale using metamorphic testing', in *Proceedings - International Conference on Software Engineering*. IEEE Press (ICSE-SEIP '21), pp. 140–149. Available at: https://doi.org/10.1109/ICSE-SEIP52600.2021.00023.

Ahmad, A., Leifler, O. and Sandahl, K. (2021) 'Empirical analysis of practitioners' perceptions of test flakiness factors', *Software Testing Verification and Reliability*, 31(8), pp. 1–24. Available at: https://doi.org/10.1002/stvr.1791.

Alshammari, A. *et al.* (2021) 'FlakeFlagger: Predicting flakiness without rerunning tests', in *Proceedings - International Conference on Software Engineering*. IEEE Press (ICSE '21), pp. 1572–1584. Available at: https://doi.org/10.1109/ICSE43902.2021.00140.

Altman, N.S. (1992) 'An introduction to kernel and nearest-neighbor nonparametric regression', *American Statistician*, 46(3), pp. 175–185. Available at: https://doi.org/10.1080/00031305.1992.10475879.

Aniche, M. (2022) *Effective software testing : A developer's guide*. Manning Publications.

Apache Software Foundation (2011) *Maven Surefire Plugin*. Available at: https://maven.apache.org/surefire/maven-surefire-plugin/ (Accessed: 20 December 2023).

Apache Software Foundation (2023) *Apache Hive*. Available at: https://hive.apache.org/ (Accessed: 26 June 2024).

Basili, V.R., Caldiera, G. and Rombach, H.D. (1994) 'The goal question metric approach', *Encyclopedia of Software Engineering*, 2, pp. 528–532. Available at: https://doi.org/10.1.1.104.8626.

Beust, C. (2004) *TestNG*. Available at: https://testng.org/doc/ (Accessed: 20 December 2023).

Camara, B. *et al.* (2021) 'On the use of test smells for prediction of flaky tests', in *ACM International Conference Proceeding Series*. New York, NY, USA: Association for Computing Machinery (SAST '21), pp. 46–54. Available at: https://doi.org/10.1145/3482909.3482916.

Chen, Y. *et al.* (2023) 'Transforming Test Suites into Croissants', in *ISSTA 2023 - Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2023), pp. 1080–1092. Available at: https://doi.org/10.1145/3597926.3598119.

Cordeiro, M. *et al.* (2022) 'Shaker: A Tool for Detecting More Flaky Tests Faster', in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press (ASE '21), pp. 1281–1285. Available at: https://doi.org/10.1109/ASE51524.2021.9678918.

Cordy, M. *et al.* (2022) 'FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment', in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE '22), pp. 982–994. Available at: https://doi.org/10.1145/3510003.3510194.

Dong, Z. *et al.* (2021) 'Flaky test detection in Android via event order exploration', in *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery (ESEC/FSE 2021), pp. 367–378. Available at: https://doi.org/10.1145/3468264.3468584.

Dutta, S. *et al.* (2020) 'Detecting flaky tests in probabilistic and machine learning applications', in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2020), pp. 211–224. Available at: https://doi.org/10.1145/3395363.3397366.

Dutta, S., Shi, A. and Misailovic, S. (2021) 'FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds', in *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery (ESEC/FSE 2021), pp. 603–614. Available at: https://doi.org/10.1145/3468264.3468615.

Fallahzadeh, E. and Rigby, P.C. (2022) 'The impact of flaky tests on historical test prioritization on chrome', in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. New York, NY, USA: Association for Computing Machinery (ICSE-SEIP '22), pp. 273–282. Available at: https://doi.org/10.1145/3510457.3513038.

Fowler, M. (2011) *Eradicating Non-Determinism in Tests*, *Martin Fowler Personal Blog*. Available at: https://martinfowler.com/articles/nonDeterminism.html (Accessed: 3 January 2024).

Freund, Yoav, and L.M. (1999) ‘The alternating decision tree learning algorithm’, in *International Conference on Machine Learning*, pp. 124–133.

Gotterbarn, D., Miller, K. and Rogerson, S. (1997) ‘Software engineering code of ethics’, *Communications of the ACM*, 40(11). Available at: https://doi.org/10.1145/265684.265699.

Gruber, M. (2022) ‘Tackling Flaky Tests: Understanding the Problem and Providing Practical Solutions’, in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press (ASE ’21), pp. 1095–1097. Available at: https://doi.org/10.1109/ASE51524.2021.9678701.

Gruber, M. and Fraser, G. (2022) ‘A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It’, *Proceedings - 2022 IEEE 15th International Conference on Software Testing, Verification and Validation, ICST 2022*, pp. 82–92. Available at: https://doi.org/10.1109/ICST53961.2022.00020.

Gruber, M. and Fraser, G. (2023) ‘FlaPy: Mining Flaky Python Tests at Scale’, in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*. IEEE Press (ICSE ’23), pp. 127–131. Available at: https://doi.org/10.1109/icse-companion58688.2023.00039.

Gyori, A. *et al.* (2015) ‘Reliable testing: Detecting state-polluting tests to prevent test dependency’, *2015 International Symposium on Software Testing and Analysis, ISSTA 2015 - Proceedings*, pp. 223–233. Available at: https://doi.org/10.1145/2771783.2771793.

Hwang, C.-L. and Yoon, K. (1981) *Multiple Attribute Decision Making: Methods and Applications A State-of-the-Art Survey*. Berlin, Heidelberg: Springer Berlin Heidelberg. Available at: https://doi.org/10.1007/978-3-642-48318-9.

Jost, S. and Tintillier, V. (2023) *Flaky Test Handler*. Available at: https://plugins.jenkins.io/flaky-test-handler/ (Accessed: 20 December 2023).

Klearman, L. (2023) *pytest-rerunfailures*. Available at: https://pypi.org/project/pytest-rerunfailures/ (Accessed: 20 December 2023).

Kowalczyk, E. *et al.* (2020) ‘Modeling and ranking flaky tests at apple’, in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE-SEIP ’20), pp. 110–119. Available at: https://doi.org/10.1145/3377813.3381370.

Lam, W. *et al.* (2019) ‘IDFlakies: A framework for detecting and partially classifying flaky tests’, *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322. Available at: https://doi.org/10.1109/ICST.2019.00038.

Lam, W., Winter, S., *et al.* (2020) ‘A large-scale longitudinal study of flaky tests’, *Proceedings of the ACM on Programming Languages*, 4(OOPSLA). Available at: https://doi.org/10.1145/3428270.

Lam, W., Muslu, K., *et al.* (2020) 'A study on the lifecycle of flaky tests', in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE '20), pp. 1471–1482. Available at: https://doi.org/10.1145/3377811.3381749.

Lam, W. (2020) *IDoFT*. Available at: https://github.com/TestingResearchIllinois/idoft (Accessed: 21 December 2023).

Lampel, J. *et al.* (2021) 'When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla', in *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery (ESEC/FSE 2021), pp. 1381–1392. Available at: https://doi.org/10.1145/3468264.3473931.

Leesatapornwongsa, T., Ren, X. and Nath, S. (2022) 'FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests', in *ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery (ESEC/FSE 2022), pp. 1509–1520. Available at: https://doi.org/10.1145/3540250.3558956.

Leong, C. *et al.* (2019) 'Assessing Transition-Based Test Selection Algorithms at Google', *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, pp. 101–110. Available at: https://doi.org/10.1109/ICSE-SEIP.2019.00019.

Li, C. *et al.* (2022a) 'Repairing Order-Dependent Flaky Tests via Test Generation', in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE '22), pp. 1881–1892. Available at: https://doi.org/10.1145/3510003.3510173.

Li, C. *et al.* (2022b) 'Repairing Order-Dependent Flaky Tests via Test Generation', in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE '22), pp. 1881–1892. Available at: https://doi.org/10.1145/3510003.3510173.

Li, C. *et al.* (2023) 'Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests', in *ISSTA 2023 - Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2023), pp. 627–638. Available at: https://doi.org/10.1145/3597926.3598083.

Li, C. and Shi, A. (2022) 'Evolution-Aware detection of order-dependent flaky tests', in *ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2022), pp. 114–125. Available at: https://doi.org/10.1145/3533767.3534404.

Luo, Q. *et al.* (2014) 'An empirical analysis of flaky tests', *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 16-21-Nove, pp. 643–653. Available at: https://doi.org/10.1145/2635868.2635920.

MacDonald, J. (2014) *Systematic Approaches to a Successful Literature Review*, *Journal of the Canadian Health Libraries Association / Journal de l'Association des bibliothèques de la santé du Canada*. Available at: https://doi.org/10.5596/c13-009.

Mendes, A. (2024a) 'Docker Framework Plugin Resolution Fix'. Available at: https://github.com/UT-SE-Research/iDFlakies/pull/66 (Accessed: 26 June 2024).

Mendes, A. (2024b) *intIDRepair*. Available at: https://github.com/AlexMViver/intIDRepair (Accessed: 11 June 2024).

Micco, J. (2017) 'The State of Continuous Integration Testing Google', *ICST 2017 - Proceedings of the 10th IEEE International Conference on Software Testing, Verification, and Validation* [Preprint].

Mudduluru, R. *et al.* (2021) 'Verifying determinism in sequential programs', in *Proceedings - International Conference on Software Engineering*. IEEE Press (ICSE '21), pp. 37–49. Available at: https://doi.org/10.1109/ICSE43902.2021.00017.

Noble, W.S. (2006) 'What is a support vector machine?', *Nature Biotechnology*, 24(12), pp. 1565–1567. Available at: https://doi.org/10.1038/nbt1206-1565.

Page, M.J. *et al.* (2021) 'The PRISMA 2020 statement: An updated guideline for reporting systematic reviews', *The BMJ*, 372. Available at: https://doi.org/10.1136/bmj.n71.

Parry, O. *et al.* (2021) 'A survey of flaky tests', *ACM Transactions on Software Engineering and Methodology*, 31(1). Available at: https://doi.org/10.1145/3476105.

Parry, O., Kapfhammer, G.M., *et al.* (2022) 'Surveying the developer experience of flaky tests', in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. New York, NY, USA: Association for Computing Machinery (ICSE-SEIP '22), pp. 253–262. Available at: https://doi.org/10.1145/3510457.3513037.

Parry, O., Hilton, M., *et al.* (2022) 'What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?', in *Proceedings - 3rd ACM/IEEE International Conference on Automation of Software Test, AST 2022*. New York, NY, USA: Association for Computing Machinery (AST '22), pp. 160–164. Available at: https://doi.org/10.1145/3524481.3527227.

Parry, O. *et al.* (2023) 'Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models', *Empirical Software Engineering*, 28(3), pp. 1–52. Available at: https://doi.org/10.1007/s10664-023-10307-w.

Pinto, G. *et al.* (2020) 'What is the Vocabulary of Flaky Tests?', in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*. New York, NY, USA: Association for Computing Machinery (MSR '20), pp. 492–502. Available at: https://doi.org/10.1145/3379597.3387482.

Polytechnic Institute of Porto (2020) 'Despacho n.º 11171/2020', pp. 193–203.

Pontillo, V., Palomba, F. and Ferrucci, F. (2021) 'Toward static test flakiness prediction: A feasibility study', in *MaLTESQuE 2021 - Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, co-located with ESEC/FSE 2021*. New York, NY, USA: Association for Computing Machinery (MaLTESQuE 2021), pp. 19–24. Available at: https://doi.org/10.1145/3472674.3473981.

Pontillo, V., Palomba, F. and Ferrucci, F. (2022) 'Static test flakiness prediction: How Far Can We Go?', *Empirical Software Engineering*, 27(7), pp. 1–57. Available at: https://doi.org/10.1007/s10664-022-10227-1.

Randoop: Automatic unit test generation for Java (2024). Available at: https://randoop.github.io/randoop/ (Accessed: 21 December 2023).

rRajivramachandran (2023) *IdFlakies gets stuck on certain repos*. Available at: https://github.com/UT-SE-Research/iDFlakies/issues/64 (Accessed: 26 June 2024).

Shi, A. *et al.* (2016) 'Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications', *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pp. 80–90. Available at: https://doi.org/10.1109/ICST.2016.40.

Shi, A. *et al.* (2019) 'IFixFlakies: A framework for automatically fixing order-dependent flaky tests', *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 545–555. Available at: https://doi.org/10.1145/3338906.3338925.

Sousa, É., Bezerra, C. and Machado, I. (2023) 'Flaky Tests in UI: Understanding Causes and Applying Correction Strategies', in *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery (SBES '23), pp. 398–406. Available at: https://doi.org/10.1145/3613372.3613406.

Strandberg, P.E. *et al.* (2020) 'Intermittently failing tests in the embedded systems domain', in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery (ISSTA 2020), pp. 337–348. Available at: https://doi.org/10.1145/3395363.3397359.

Tahir, A. *et al.* (2023) 'Test flakiness' causes, detection, impact and responses: A multivocal review', *Journal of Systems and Software*, 206, p. 111837. Available at: https://doi.org/10.1016/j.jss.2023.111837.

Vahabzadeh, A., Fard, A.M. and Mesbah, A. (2015) 'An empirical study of bugs in test code', *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pp. 101–110. Available at: https://doi.org/10.1109/ICSM.2015.7332456.

Verdecchia, R. *et al.* (2021) 'Know You Neighbor: Fast Static Prediction of Test Flakiness', *IEEE Access*, 9, pp. 76119–76134. Available at: https://doi.org/10.1109/ACCESS.2021.3082424.

Wang, R., Chen, Y. and Lam, W. (2022) 'IPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests', in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery (ICSE '22), pp. 120–124. Available at: https://doi.org/10.1145/3510454.3516846.

Wei, A. *et al.* (2021) 'Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs for Detecting Order-Dependent Flaky Tests', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12651 LNCS, pp. 270–287. Available at: https://doi.org/10.1007/978-3-030-72016-2 15.

Wei, A. *et al.* (2022) 'Preempting Flaky Tests via Non-Idempotent-Outcome Tests', in *Proceedings - International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ICSE '22), pp. 1730–1742. Available at: https://doi.org/10.1145/3510003.3510170.

Wohlin, C. (2014) 'Guidelines for snowballing in systematic literature studies and a replication in software engineering', *ACM International Conference Proceeding Series* [Preprint]. Available at: https://doi.org/10.1145/2601248.2601268.

Yi, P. *et al.* (2021) 'Finding Polluter Tests Using Java PathFinder', *SIGSOFT Softw. Eng. Notes*, 46(3), pp. 37–41. Available at: https://doi.org/10.1145/3468744.3468756.

Zhang, P. *et al.* (2021) 'Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications', in *Proceedings - International Conference on Software Engineering*. IEEE Press (ICSE '21), pp. 50–61. Available at: https://doi.org/10.1109/ICSE43902.2021.00018.

Zhang, S. *et al.* (2014) 'Empirically revisiting the test independence assumption', *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pp. 385–396. Available at: https://doi.org/10.1145/2610384.2610404.

Zolfaghari, B. *et al.* (2021) 'Root causing, detecting, and fixing flaky tests: State of the art and future roadmap.', *Software: Practice & Experience*, 51(5), pp. 851–867. Available at: https://doi.org/10.1002/spe.2929.

Appendix A. GQM Measurement Metrics

Table 30 - Measurement metrics

Metric	Description
Compatibility score	Calculation of the percentage of tool compatibility between detection tools and repair tools according to various criteria addressing concerns such as the relevancy of produced data for usage by the combined tool, execution environment compatibilities, and data structuring, among others.
Execution time	Comparison of the amount of time necessary for the unified solution to execute for the given data compared to the sum of the execution time of the unified components when run separately.
Detection Recall	Calculation of the proportion of tests correctly classified as flaky compared to the total flaky tests in the dataset. This presupposes the dataset's flaky tests as the only verifiably true positives in the project. $$\frac{True\ Positives}{True\ Positives + False\ Negatives}$$
Repair Recall	Calculation of the proportion of true flaky tests repaired compared to the total tests classified as flaky by the unified approach. This presupposes the dataset's flaky tests as the only verifiably true positives in the project, any test not present in the dataset that the repair component receives, and repairs is deemed a false positive. $$\frac{True\ Positives}{True\ Positives + False\ Positives}$$
Number of execution steps	Comparison between the number of steps required to execute the unified solution and the number of steps needed to separately run the unified components.

Appendix B. Systematic Mapping Study: full data

Table 31 - Studies accepted in the full-text screening

Title	**Reference**
A large-scale longitudinal study of flaky tests	(Lam, Winter, *et al.*, 2020)
A Study on the Lifecycle of Flaky Tests	(Lam, Muslu, *et al.*, 2020)
A survey of flaky tests	(Parry *et al.*, 2021)
Detecting flaky tests in probabilistic and machine learning applications	(Dutta *et al.*, 2020)
Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications	(Zhang *et al.*, 2021)
Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models	(Parry *et al.*, 2023)
Evolution-Aware detection of order-dependent flaky tests	(Li and Shi, 2022)
FlakeFlagger: Predicting flakiness without rerunning tests	(Alshammari *et al.*, 2021)
FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests	(Leesatapornwongsa, Ren and Nath, 2022)
Flaky test detection in Android via event order exploration	(Dong *et al.*, 2021)
Flaky Tests in UI: Understanding Causes and Applying Correction Strategies	(Sousa, Bezerra and Machado, 2023)
FlaPy: Mining Flaky Python Tests at Scale	(Gruber and Fraser, 2023)
FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds	(Dutta, Shi and Misailovic, 2021)
IPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests	(Wang, Chen and Lam, 2022)
Know You Neighbor: Fast Static Prediction of Test Flakiness	(Verdecchia *et al.*, 2021)
On the use of test smells for prediction of flaky tests	(Camara *et al.*, 2021)
Preempting Flaky Tests via Non-Idempotent-Outcome Tests	(Wei *et al.*, 2022)

Title	Reference
Repairing Order-Dependent Flaky Tests via Test Generation	(Li *et al.*, 2022a)
Root causing, detecting, and fixing flaky tests: State of the art and future roadmap.	(Zolfaghari *et al.*, 2021)
Shaker: A Tool for Detecting More Flaky Tests Faster	(Cordeiro *et al.*, 2022)
Static test flakiness prediction: How Far Can We Go?	(Pontillo, Palomba and Ferrucci, 2022)
Surveying the developer experience of flaky tests	(Parry, Kapfhammer, *et al.*, 2022)
Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests	(Li *et al.*, 2023)
Test flakiness' causes, detection, impact and responses: A multivocal review	(Tahir *et al.*, 2023)
Toward static test flakiness prediction: A feasibility study	(Pontillo, Palomba and Ferrucci, 2021)
Transforming Test Suites into Croissants	(Chen *et al.*, 2023)
Verifying determinism in sequential programs	(Mudduluru *et al.*, 2021)
What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?	(Parry, Hilton, *et al.*, 2022)
What is the Vocabulary of Flaky Tests?	(Pinto *et al.*, 2020)
When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla	(Lampel *et al.*, 2021)

Table 32 - Studies removed from the full-text screening

Title	Reference	Reason for not including
Empirical analysis of practitioners' perceptions of test flakiness factors	(Ahmad, Leifler and Sandahl, 2021)	Does not fulfill any inclusion criteria
FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment	(Cordy *et al.*, 2022)	Does not fulfill any inclusion criteria
Intermittently failing tests in the embedded systems domain	(Strandberg *et al.*, 2020)	Does not fulfill any inclusion criteria
Modeling and ranking flaky tests at apple	(Kowalczyk *et al.*, 2020)	Does not fulfill any inclusion criteria
Tackling Flaky Tests: Understanding the Problem and Providing Practical Solutions	(Gruber, 2022)	Does not fulfill any inclusion criteria
Testing web enabled simulation at scale using metamorphic testing	(Ahlgren *et al.*, 2021)	Does not fulfill any inclusion criteria
The impact of flaky tests on historical test prioritization on chrome	(Fallahzadeh and Rigby, 2022)	Does not fulfill any inclusion criteria

Table 33 - Studies added via snowballing

Title	Reference
Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications	(Shi *et al.*, 2016)
IFixFlakies: A framework for automatically fixing order-dependent flaky tests	(Cordy *et al.*, 2022) (Shi *et al.*, 2019)
Empirically revisiting the test independence assumption	(Zhang *et al.*, 2014)
IDFlakies: A framework for detecting and partially classifying flaky tests	(Lam *et al.*, 2019)
Finding Polluter Tests Using Java PathFinder	(Yi *et al.*, 2021)

Appendix C. Experiment Dataset

Table 34 - Experiment Dataset Projects

Project	OD tests	NOD tests
spring-cloud-netflix	10	0
json-iterator	20	8
nifi	21	0
fastjson	15	27
wasp	28	115
hive	19	0
spring-cloud-kubernetes	30	0
visualee	47	0
ormlite-core	87	0
remoting	10	7

Appendix D. Compatibility Assessment Criteria

Table 35 - Values for content relevance criteria

Criteria Evaluation	Description
4. Excellent	The detection output contains all necessary repair input data fields with clear identification of each data field.
3. Good	The detection output contains all necessary repair input data fields, but some relevant details are ambiguous or missing.
2. Medium	The detection output contains some relevant data fields but significant details necessary for the repair input data are missing or unclear.
1. Poor	The detection output lacks most, or all the relevant information required for repair, making it ineffective or unusable.

Table 36 - Values for file Structure compatibility criteria

Criteria Evaluation	Description
4. Excellent	The detection output exactly matches the expected file structure of the repair input data. This includes folder organization and appropriate file extensions for the files containing relevant data fields.
3. Good	The detection output closely resembles the expected file structure of the repair tool's input data. While the general folder organization is similar, there may be minor inconsistencies or deviations. File extensions corresponding to the repair input data for relevant data fields may be present, but there are occasional discrepancies.
2. Medium	The detection output partially resembles the expected file structure of the repair tool's input data. Significant differences can be identified in folder organization, and various file extensions for relevant data fields may be inconsistent with the repair tool input data. The overall structure may require adjustments to align with the repair tool's input requirements.
1. Poor	The detection output does not match the expected file structure of the repair tool's input data. Folder organization is entirely different and file extensions for the relevant data fields do not match the repair input data. The structure is incompatible with the repair tool's input requirements, requiring substantial modifications for compatibility.

Table 37 - Values for data representation consistency criteria

Criteria Evaluation	Description
4. Excellent	The detection output exhibits perfect consistency with the data representation expected by the repair input. Data types, data structures, and the organization of the data align precisely, with no discrepancies or inconsistencies. The format of data fields and hierarchical organization is identical.
3. Good	The detection output exhibits good consistency with the data representation expected by the repair input. While there may be minor differences or deviations in data types, structures, or organization, these discrepancies do not significantly impact compatibility. The overall format of data fields and hierarchical organization remains largely consistent, allowing for effective interoperability between the detection and repair tools with minor adjustments.
2. Medium	The detection output exhibits fair consistency with the data representation expected by the repair input. There are noticeable differences or inconsistencies in data types, structures, or organization, requiring moderate adjustments to ensure compatibility. The format of data fields or hierarchical organization may vary to some extent, potentially hindering seamless interoperability between the detection and repair tools.
1. Poor	The detection output lacks consistency with the data representation expected by the repair input. Significant differences or discrepancies in data types, structures, or organization make interoperability between the detection and repair tools challenging. The format of data fields or hierarchical organization is entirely incompatible, necessitating substantial modifications for effective integration between the detection and repair tools.

Table 38 - Values for execution environment compatibility criteria

Criteria Evaluation	Description
4. Excellent	The execution environments required by the detection and repair tools are fully compatible with each other, both tools can run seamlessly in the same environment without any conflicts or compatibility issues. Operating systems, hardware configurations, and other environmental factors align perfectly.
3. Good	The execution environments required by the detection and repair tools are mostly compatible with each other. Minor differences or constraints may exist between the environments, but they do not affect compatibility or functionality. With minor adjustments or configurations, both tools can run effectively in the same environment, allowing for successful integration and operation.
2. Medium	The execution environments required by the detection and repair tools have moderate compatibility issues. There are noticeable differences or constraints between the environments, requiring moderate adjustments or configurations for compatibility. Despite these challenges, the tools can potentially still run in the same environment, although integration may be less straightforward.
1. Poor	The execution environments required by the detection and repair tools are incompatible with each other. Significant differences or constraints exist between the environments, preventing successful integration and operation. Running both tools in the same environment is impractical or impossible without extensive modifications or workarounds.

Table 39 - Dependency version compatibility criterium evaluations

Criteria Evaluation	Description
4. Excellent	The versions of dependencies required by the detection and repair tools are fully compatible with each other. Both tools can seamlessly utilize the same versions of dependencies without conflicts or compatibility issues.
3. Good	The versions of dependencies required by the detection and repair tools are mostly compatible with each other. Minor differences in version requirements may exist but they do not significantly affect compatibility or functionality. With minor adjustments or updates, both tools can utilize compatible versions of their dependencies.
2. Medium	The versions of dependencies required by the detection and repair tools have moderate compatibility issues. Noticeable differences in version requirements exist, requiring moderate adjustments or updates for compatibility. While difficult, the tools could be adjusted to achieve dependence compatibility.
1. Poor	The versions of dependencies required by the detection and repair tools are incompatible with each other. Significant differences in version requirements prevent successful integration of the tools. Identifying compatible versions of dependencies is impractical or impossible without extensive modifications or workarounds, making integration unfeasible.

Appendix E. Tool Evaluation Execution Results

Table 40 - iDFlakies Dataset Execution Results

Project	OD tests in dataset	NOD tests in dataset	Detected OD tests	Detected NOD tests	Other detected tests	Recall (OD)	Recall (NOD)	Recall (All tests)	Time (mins)
spring-cloud-netflix	10	0	8	0	1	0,8	-	0,8	44
json-iterator	20	8	20	1	0	1	0,13	0,75	9
nifi	21	0	0	0	17	0	-	0	494
fastjson	15	27	0	0	0	0	0	0	12
wasp	28	115	28	89	3	1	0,77	0,82	215
hive	19	0	2	0	1	0,16	-	0,16	789
spring-cloud-kubernetes	30	0	30	0	26	1	-	1	481
visualee	47	0	47	0	0	1	-	1	10
ormlite-core	87	0	87	0	0	1	-	1	61
remoting	10	7	0	0	0	0	0	0	380

Table 41 - Nondex Dataset Execution Results

Project	OD tests in dataset	NOD tests in dataset	Detected OD tests	Detected NOD tests	Other detected tests	Recall (OD)	Recall (NOD)	Recall (All tests)	Time taken (mins)
spring-cloud-netflix	10	0	0	0	0	0	-	0	13
json-iterator	20	8	1	8	18	0,05	1	0,32	1
nifi	21	0	0	0	133	0	-	0	19
fastjson	15	27	0	19	26	0	0,70	0,45	4
wasp	28	115	0	7	59	0	0,06	0,05	2
hive	19	0	0	0	1390	0	-	0	1366
spring-cloud-kubernetes	30	0	13	0	74	0,43	-	0,43	40
visualee	47	0	1	0	0	0,02	-	0,02	0
ormlite-core	87	0				0	-	0	0
remoting	10	7	0	0	0	0	0	0	9

Table 42 - Shaker Dataset Execution Results

Project	OD tests in dataset	NOD tests in dataset	Detected OD tests	Detected NOD tests	Other detected tests	Recall (OD)	Recall (NOD)	Recall (All tests)	Time taken (mins)
spring-cloud-netflix	10	0	0	0	0	0	-	0	21
json-iterator	20	8	1	8	0	0,05	1	0,32	1
nifi	21	0	0	0	0	0	-	0	216
fastjson	15	27	0	11	5	0	0,41	0,26	9
wasp	28	115	0	1	3	0	0,01	0,01	3
hive	19	0	0	0	0	0	-	0	Interrupted after 24 hours
spring-cloud-kubernetes	30	0	0	0	0	0	-	0	101
visualee	47	0	1	0	0	0,02	-	0,02	1
ormlite-core	87	0	0	0	0	0	-	0	1
remoting	10	7	0	0	0	0	0	0	26

Table 43 - intIDRepair Dataset Execution Results (Complete overview)

Project	Detected OD tests	Detected NOD tests	Other detected tests	Detection Recall (OD)	Detection Recall (NOD)	Detection Recall (All tests)	Gen. Patches (OD)	Repair Recall (Dataset tests)	Repair Recall (Detected tests)	Time taken (mins)
spring-cloud-netflix	8	0	1	0,8	-	0,8	0	0	0	39
json-iterator	20	1	0	1	0,13	0,75	0	0	0	17
nifi	1	0	3	0,05	-	0,048	0	0	0	174
fastjson	0	0	0	0	0	0	0	0	-	112
wasp	28	89	3	1	0,77	0,82	0	0	0	226
hive	2	0	1	0,11	-	0,11	0	0	0	792
spring-cloud-kubernetes	29	0	26	0,97	-	0,97	0	0	0	456
visualee	47	0	0	1	-	1	45	0,96	0,96	227
ormlite-core	87	0	0	1	-	1	0	0	0	76
remoting	0	0	0	0	0	0	0	0	-	558