



Understanding and Mitigating Flaky Software Test Cases

Owain Ben Parry

The University of Sheffield

Faculty of Engineering
Department of Computer Science

June 2023

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

Abstract

A *flaky test* is a test case that can pass or fail without changes to the test case code or the code under test. They are a wide-spread problem with serious consequences for developers and researchers alike. For developers, flaky tests lead to time wasted debugging spurious failures, tempting them to ignore future failures. While unreliable, flaky tests can still indicate genuine issues in the code under test, so ignoring them can lead to bugs being missed. The non-deterministic behaviour of flaky tests is also a major snag to continuous integration, where a single flaky test can fail an entire build. For researchers, flaky tests challenge the assumption that a test failure implies a bug, an assumption that many fundamental techniques in software engineering research rely upon, including test acceleration, mutation testing, and fault localisation.

Despite increasing research interest in the topic, open problems remain. In particular, there has been relatively little attention paid to the views and experiences of developers, despite a considerable body of empirical work. This is essential to guide the focus of research into areas that are most likely to be beneficial to the software engineering industry. Furthermore, previous automated techniques for detecting flaky tests are typically either based on exhaustively rerunning test cases or machine learning classifiers. The prohibitive runtime of the rerunning approach and the demonstrably poor inter-project generalisability of classifiers leaves practitioners with a stark choice when it comes to automatically detecting flaky tests.

In response to these challenges, I set two high-level goals for this thesis: (1) to enhance the understanding of the manifestation, causes, and impacts of flaky tests; and (2) to develop and empirically evaluate efficient automated techniques for mitigating flaky tests. In pursuit of these goals, this thesis makes five contributions: (1) a comprehensive systematic literature review of 76 published papers; (2) a literature-guided survey of 170 professional software developers; (3) a new feature set for encoding test cases in machine learning-based flaky test detection; (4) a novel approach for reducing the time cost of rerunning-based techniques for detecting flaky tests by combining them with machine learning classifiers; and (5) an automated technique that detects and classifies existing flaky tests in a project and produces reusable project-specific machine learning classifiers able to provide fast and accurate predictions for future test cases in that project.

Acknowledgements

First and foremost I would like to thank my supervisor Phil McMinn and my two collaborators Gregory M. Kapfhammer and Michael Hilton. I would also like to thank the following past and present students, researchers, and academics (in no particular order) from the verification and testing lab at the University of Sheffield: Ben, Ibrahim, Islam, Torrell, Megan, Firhard, Abdullah, George (x2), Dave, Yasmeen, Tom, Panayiotis, Richard, Nina, Michael, Andy, Giulia, Neil, and José. Finally I would like to thank Caitlin, Lewis, Soffia, Jack, Henry, and all my other friends, my parents John and Louise, and the rest of my family.

Publications

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Flake it ‘till you make it: Using automated repair to induce and fix latent test flakiness. In *Proceedings of the International Workshop on Automated Program Repair (APR)*, pages 11–12, 2020

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. A survey of flaky tests. *Transactions on Software Engineering and Methodology*, 31(1):1–74, 2021

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 93–104, 2022

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 253–262, 2022

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 160–164, 2022

O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(72), 2023

Contents

Abstract	i
Acknowledgements	ii
Publications	iii
1 Introduction	1
1.1 Goals	2
1.2 Contributions	2
1.2.1 Systematic Literature Review (Chapter 2)	3
1.2.2 Developer Survey (Chapter 3)	3
1.2.3 FLAKE16 (Chapter 4)	4
1.2.4 CANNIER (Chapter 5)	4
1.2.5 FLAKEFRIEND (Chapter 6)	5
2 A Survey of Flaky Tests	6
2.1 Introduction	6
2.2 Methodology	7
2.2.1 Survey Scope	8
2.2.2 Collection Approach	8
2.2.3 Collection Results	9
2.3 Causes and Associated Factors	10
2.3.1 General Causes	12
2.3.2 Test Order Dependencies	16
2.3.3 Application-Specific Causes	19
2.4 Costs and Consequences	22
2.4.1 Consequences for Testing Reliability	24
2.4.2 Consequences for the Efficiency of Testing	25
2.4.3 Other Consequences	29
2.5 Detection	32
2.5.1 Detecting Flaky Tests	34
2.5.2 Detecting Order-Dependent Tests	42
2.6 Mitigation and Repair	50
2.6.1 Mitigation	52
2.6.2 Repair	56
2.7 Analysis	63
2.7.1 Prominent Papers and Authors	63
2.7.2 Tools and Subject Sets	64
2.7.3 Related Research Areas	66
2.7.4 Limitations and Future Directions	67
2.8 April 2023 Update	69
2.8.1 RQ1: What are the causes and associated factors of flaky tests?	69

2.8.2	RQ2: What are the costs and consequences of flaky tests?	71
2.8.3	RQ3: What insights and techniques can be applied to detect flaky tests? . .	72
2.8.4	RQ4: What insights and techniques can be applied to mitigate or repair flaky tests?	76
2.9	Conclusion	80
3	Surveying the Developer Experience of Flaky Tests	82
3.1	Introduction	82
3.2	Methodology	83
3.2.1	Developer Survey	83
3.2.2	StackOverflow Threads	86
3.2.3	Analysis	86
3.2.4	Threats to Validity	87
3.3	Results	87
3.3.1	RQ1: Definition	88
3.3.2	RQ2: Impacts	89
3.3.3	RQ3: Causes	90
3.3.4	RQ4: Actions	93
3.4	Recommendations	94
3.5	Related Work	96
3.6	Conclusion	96
4	Evaluating Features for Machine Learning Detection of Order- and Non-Order- Dependent Flaky Tests	97
4.1	Introduction	97
4.2	The FLAKE16 Feature Set	98
4.3	Evaluation	100
4.3.1	Data Collection	100
4.3.2	Data Preprocessing	101
4.3.3	Data Balancing	102
4.3.4	Machine Learning Classifiers	102
4.3.5	Methodology	103
4.3.6	Threats to Validity	104
4.4	Empirical Results	105
4.4.1	RQ1. Compared to the features used by FLAKEFLAGGER, does the FLAKE16 feature set improve the performance of flaky test case detection with machine learning classifiers?	105
4.4.2	RQ2. Can machine learning classifiers effectively detect order-dependent flaky test cases?	106
4.4.3	RQ3. Which features of FLAKE16 are the most impactful?	106
4.5	Discussion	107
4.5.1	General Classifier Performance	107
4.5.2	Reliability of Performance Metrics	109
4.5.3	Impact of Features	109
4.5.4	Impact of Preprocessing, Balancing, and Classifier Choice	110
4.5.5	Implications	110
4.6	Related Work	110
4.7	Conclusion	111
5	Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models	112
5.1	Introduction	112
5.2	Background	113
5.2.1	RERUN	113

5.2.2	IDFLAKIES	113
5.2.3	PAIRWISE	114
5.3	The CANNIER Approach	114
5.3.1	Motivating Example	114
5.3.2	Single-Classifier CANNIER	116
5.3.3	Multi-Classifier CANNIER	116
5.4	Tooling	117
5.4.1	pytest-CANNIER	117
5.4.2	CANNIER-Framework	118
5.5	Empirical Evaluation	124
5.5.1	Subject Set	125
5.5.2	Methodology	125
5.5.3	Threats to Validity	130
5.6	Results	132
5.6.1	RQ1. How effective is machine learning-based flaky test detection?	132
5.6.2	RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?	135
5.6.3	RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?	137
5.6.4	RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?	139
5.7	Discussion	144
5.7.1	RQ1. How effective is machine learning-based flaky test detection?	144
5.7.2	RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?	145
5.7.3	RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?	145
5.7.4	RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?	147
5.7.5	Implications	149
5.8	Related Work	149
5.9	Conclusions	150
6	Automatically Training Project-Specific Machine Learning Models to Detect and Classify Flaky Tests	152
6.1	Introduction	152
6.2	FLAKEFRIEND	153
6.2.1	Feature Extraction	154
6.2.2	Project-Agnostic Prediction	154
6.2.3	Project-Agnostic Oracle	155
6.2.4	Project-Specific Training	155
6.2.5	Project-Specific Prediction	155
6.3	Implementation	155
6.3.1	Feature Extraction	156
6.3.2	Project-Agnostic Prediction	157
6.3.3	Project-Agnostic Oracle	157
6.3.4	Project-Specific Training	158
6.3.5	Project-Specific Prediction	159
6.4	Evaluation	159
6.4.1	Subjects	159
6.4.2	RQ1. What is the detection performance and time cost of FLAKEFRIEND for existing test cases?	159
6.4.3	RQ2. What is the detection performance of FLAKEFRIEND for future test cases?	160

6.4.4	RQ3. To what extent does FLAKEFRIEND reduce cumulative time cost as the test suite grows?	160
6.4.5	Threats to Validity	161
6.5	Results	161
6.5.1	RQ1. What is the detection performance and time cost of FLAKEFRIEND for existing test cases?	162
6.5.2	RQ2. What is the detection performance of FLAKEFRIEND for future test cases?	163
6.5.3	RQ3. To what extent does FLAKEFRIEND reduce cumulative time cost as the test suite grows?	163
6.6	Discussion	164
6.6.1	Detection Performance	164
6.6.2	Time Cost	165
6.6.3	Implications for Developers	165
6.7	Related Work	166
6.8	Conclusion	166
7	Conclusion and Future Work	168
7.1	Restrictions and Limitations	168
7.1.1	Types of Subject Programs	168
7.1.2	Empirical Uncertainty	170
7.2	Future Work	170
7.2.1	Extended Replication	170
7.2.2	Extended Developer Survey	170
7.2.3	Automatically Generated Flakiness	170
7.2.4	Latent Flakiness	171
7.2.5	Automated Explanation	171
A	Appendix A: Ethics Application	189

Chapter 1

Introduction

“Don’t cry ‘wolf’, shepherd boy,” said the villagers, “when there’s no wolf!” - “The Boy Who Cried Wolf”, Aesop

Software developers rely on test cases to identify bugs in their code and to provide a signal as to their code’s correctness [77]. Should such signals have a history of unreliability, they not only become less informative, but may also be considered untrustworthy [180, 181]. In the context of software testing, practitioners refer to these unreliable signals as *flaky tests*. The definition varies slightly, but a flaky test is generally defined as a test case that can pass and fail without changes to the test case code or the code under test, such as Figure 1.1. Concurrency and randomness are well-established causes among many others [37, 59, 89, 105, 155], though flakiness has far-reaching negative consequences regardless of origin. These consequences are felt by developers from small open-source projects to the likes of Google, Microsoft, and Meta [88, 106, 110].

Flaky tests challenge the assumption that a test failure implies a bug, constituting a leading cause of “false alarm” test failures, and potentially more seriously, having the potential to mask the presence of a genuine bug [155, 175]. Naturally, this poses serious problems for developers and researchers alike. For developers, flaky tests may lead to time wasted debugging spurious failures, leading them to ignore future test failures. This is detrimental to software stability, because while a flaky test may be unreliable, it could still indicate a genuine bug in some instances [134, 181]. This is further exacerbated when flaky tests accumulate, as developers may lose trust in the entire test suite [180]. The non-deterministic behaviour of flaky tests is also a serious hindrance to continuous integration (CI). A study focused on the Travis CI platform found that 47% of failing builds that were manually restarted eventually passed without any changes, indicating the presence of flaky tests [33]. Flaky tests are not a rare phenomenon, with 20% of developers claiming to experience them monthly, 24% encountering them on a weekly basis, and 15% dealing with them daily [37].

Flaky tests are a threat to the validity of any methodology that assumes test case outcomes are deterministic, making them an obstacle to the deployment of many techniques in software engineering research. A specific category, known as *order-dependent* (OD) flaky tests, are dependent on the test execution order. This means that techniques for accelerating the testing process, including test case prioritisation, selection, and parallelisation, are unsound for test suites containing OD flaky tests [91]. Other techniques negatively impacted by flaky tests include fault localisation [25, 157], mutation testing [68, 139], and automatic test suite generation [10, 125, 138].

Given the pain caused by flaky tests, interest in the topic has been increasing over the last ten years. This has culminated in many empirical studies on their presentation and causes [59, 69, 105, 136, 155, 175]. It has also resulted in automated techniques for the detection, debugging, and repair of flaky tests [8, 16, 64, 90, 133, 141]. Despite valuable advances, there remains open problems. Particularly in the past five years, the number of papers published on or related to flaky tests has soared. Zolfaghari et al. [179] provided a literature review on some of the main techniques for mitigating flaky tests. However, to date there has been no comprehensive survey that covers not only this core literature but also the wider periphery, such as the influence that flaky tests have in the wider software engineering field. A complete systematic literature review at this time

```

1  def test_spinup_time(hook):
2      data = []
3      for i in range(10000):
4          data.append(torch.Tensor(5, 5).random_(100))
5          start_time = time()
6          dummy = sy.VirtualWorker(hook, id="dummy", data=data)
7          end_time = time()
8      assert (end_time - start_time) < 1

```

Figure 1.1: A flaky test from PySyft [259]. It contains an assertion on line 8 regarding the time taken to complete an operation. Because the time taken may vary depending on the machine specification, it can fail regardless of any bugs. This test was so flaky that the developers decided to remove it [269].

of burgeoning interest is long overdue and is essential for existing researchers to identify areas of future work and for new researchers to get up to speed with the state of the art. Furthermore, there is little focus on the views and experiences of developers, despite a considerable body of empirical work. Since flaky tests are primarily a developer problem, there is an underutilised opportunity to acquire valuable insights from those who experience them first-hand. This is essential to guide the focus of research into areas that are most likely to be beneficial to the software engineering industry. Where previous studies do exist, they focus on specific organisations and developers' self-reported experiences [37, 71], two potential sources of bias [31].

Given the problems associated with flaky tests, researchers introduced automated techniques to detect them. Many involve a significant number of repeated test executions and some require instrumentation [15, 17, 35, 47, 90, 144, 175], making them prohibitively expensive for deployment in large software projects. This motivated researchers to develop techniques based on machine learning classifiers trained using static features of test cases, such as their length, complexity, and the presence of particular keywords and identifiers [128, 159]. One study found that combining static features with dynamically-collected characteristics, like execution time and line coverage, resulted in better detection performance at the relatively minimal cost of a single, instrumented test suite run [8]. Prior studies have only evaluated a limited set of features, while the broader literature has identified many more test case characteristics that may be indicative of flakiness. Without further evaluation, machine learning classifiers cannot be used to their full potential for detecting flaky tests. While previous evaluations based on cross-validation show promising results, researchers have established that classifiers for detecting flaky tests perform poorly when evaluated on test cases from projects that were not part of their training data [20, 41]. Therefore, the prohibitive time cost of rerunning-based techniques and the limited performance and potential lack of inter-project generalisability of machine learning-based techniques leaves practitioners with a stark choice when it comes to automatically detecting flaky tests in their projects.

1.1 Goals

In response to these open challenges, I set two high-level goals for this thesis:

1. **Understanding:** To enhance the understanding of the manifestation, causes, and impacts of flaky tests through literature review, developer survey, and empirical evaluation.
2. **Mitigating:** To develop and empirically evaluate efficient automated techniques for mitigating the problem of flaky tests through accurate and practical detection.

1.2 Contributions

In pursuit of these goals, this thesis makes five main contributions. The following subsections summarise the chapters associated with each of them (see Figure 1.2).

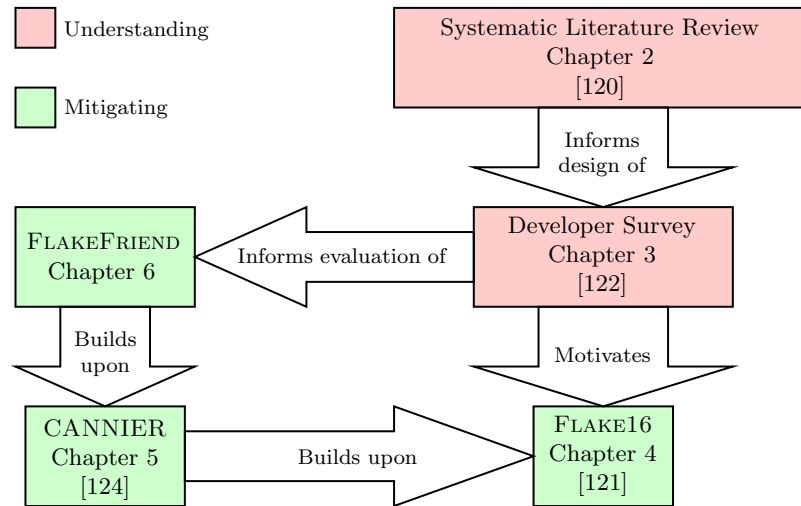


Figure 1.2: The relationships between the chapters of this thesis. Citations indicate that a chapter is based on my published work. Red blocks address “understanding” and green blocks address “mitigating”.

1.2.1 Systematic Literature Review (Chapter 2)

A comprehensive survey of the body of literature relevant to flaky test research, covering 76 papers. I split the analysis into four parts: addressing the causes of flaky tests, their costs and consequences, detection strategies, and approaches for their mitigation and repair. The findings and their implications have consequences for how the software testing community deals with test flakiness, pertinent to practitioners and of interest to those wanting to familiarise themselves with the research area. The review makes the following contributions:

1. **Evaluation (Section 2.2)**: An evaluation of the growing research area of flaky tests.
2. **Causes (Section 2.3)**: A comparative analysis on the causes of flaky tests.
3. **Costs (Section 2.4)**: A summary of costs and impacts on testing reliability and efficiency.
4. **Detection (Section 2.5)**: A comparison of tools and strategies for detecting flaky tests.
5. **Mitigation (Section 2.6)**: An analysis of techniques to mitigate and repair flaky tests.
6. **Discussion (Section 2.7)**: A discussion of research threads, trends, and future directions.

1.2.2 Developer Survey (Chapter 3)

A developer survey that received 170 responses, the design of which was closely guided by the findings in Chapter 2. Having previously examined flaky tests through the lens of published literature, this study characterises the experience of flaky tests by directly consulting developers. I also searched on StackOverflow and analysed 38 threads relevant to flaky tests, offering a distinct perspective free of any self-reporting bias. Using a mixture of numerical and thematic analyses, I made a number of findings, including (1) developers strongly agree that flaky tests hinder continuous integration; (2) developers who experience flaky tests more often may be more likely to ignore potentially genuine test failures; and (3) developers rate issues in setup and teardown to be the most common causes of flaky tests. The survey makes the following contributions:

1. **Developer Survey (Section 3.2.1)**: I designed a survey based on previous literature and received 170 responses. Through numerical and thematic analysis, I identified alternative definitions of flaky tests, the most significant impacts of flaky tests, the most frequent causes of flaky tests, and the most common actions developers perform in response to flaky tests.

2. **StackOverflow Threads (Section 3.2.2)**: I procured a dataset of 38 StackOverflow threads and through thematic analysis I offer a unique insight into the causes of flaky tests experienced by developers and the strategies that they suggest to repair them, independent of what they self-reported in the developer survey.
3. **Findings and Recommendations (Section 3.4)**: I surfaced a range of findings that support previous literature and some that were more unforeseen. From these, I offer actionable recommendations for both software developers and researchers.

1.2.3 FLAKE16 (Chapter 4)

A new feature set for encoding test cases in machine learning-based flaky test detection. Having observed an increasing trend of researchers applying machine learning classifiers to detecting flaky tests in Chapter 2, and having found that developers rate improper setup and teardown as the most common cause of flaky tests in Chapter 3, this chapter fills a valuable research gap by considering the detection of OD flaky tests with machine learning. Using 54 distinct pipelines of data preprocessing, data balancing, and machine learning classifiers for detecting both non-order-dependent (NOD) and OD flaky tests, I compared FLAKE16 to a previous feature set. I used the test suites of 26 Python projects as subjects, consisting of over 67,000 test cases. Along with identifying the most impactful metrics when detecting both types of flaky test, the empirical study shows how FLAKE16 is better than prior work, including (1) a 13% increase in overall F1 score when detecting NOD flaky tests and (2) a 17% increase in overall F1 score when detecting OD flaky tests. The study makes the following contributions:

1. **Feature Set (Section 4.2)**: A new feature set for machine learning-based flaky test detection, FLAKE16. The evaluation demonstrated an improved detection performance for both NOD and OD flaky tests compared to a previous feature set.
2. **Evaluation (Section 4.3)**: My evaluation of 54 machine learning pipelines is the first to consider the detection of OD flaky tests, offering a more complete assessment of the applicability of machine learning to the problem of flaky test detection.
3. **Findings and Implications (Sections 4.4 and 4.5)**: Leveraging the empirical results, the study surfaced findings with implications relevant to both the research community and software developers, including the most impactful test case metrics for detecting flaky tests.
4. **Dataset**: To collect the data required to perform the experiments, I developed a comprehensive framework of tools, `Flake16Framework`. To identify flaky tests, I used `Flake16Framework` to execute 5,000 times the test suites for 26 programs containing over 67,000 test cases. Supporting the replication of this study's results and further investigations into machine learning for flaky test detection, I make `Flake16Framework` and all of the data available in the replication package [218].

1.2.4 CANNIER (Chapter 5)

A novel approach for reducing the time cost of rerunning-based techniques for detecting flaky tests by combining them with machine learning classifiers. Despite having observed a definite improvement in machine learning-based detection using FLAKE16 in Chapter 4, the detection performance of machine learning classifiers was still fairly lacklustre. My empirical evaluation involving 30 Python projects demonstrated that CANNIER can reduce the time cost of existing rerunning-based techniques by an order of magnitude while maintaining a detection performance that is significantly better than classifiers alone. Furthermore, the comprehensive study extends existing work on machine learning-based detection and revealed a number of additional findings, including (1) the performance of machine learning classifiers for detecting polluter test cases (a type of test case involved in establishing an OD flaky test [141]); (2) using the mean values of dynamic test case features from repeated measurements can slightly improve the detection performance

of machine learning classifiers; and (3) correlations between various test case features and the probability of the test case being flaky. The study makes the following contributions:

1. **Approach (Section 5.3):** CANNIER significantly reduces the time cost of rerunning-based flaky test detection with a minimal decrease in detection performance.
2. **Tooling (Section 5.4):** To facilitate the empirical evaluation and allow for replication, I developed a framework of automated tools that I make freely available [223].
3. **Evaluation (Section 5.5):** A comprehensive empirical evaluation demonstrated the effectiveness of CANNIER’s combination of rerunning and machine learning techniques, revealing further novel findings about machine learning-based flaky test detection, such as the performance of machine learning classifiers for detecting polluter test cases.
4. **Dataset (Section 5.5.1):** A dataset containing 89,668 tests from 30 Python projects taking over six weeks of compute time to produce. I make this available as part of the replication package to enable further research [217].

1.2.5 FLAKEFRIEND (Chapter 6)

An automated technique that detects and classifies existing flaky tests in a project, and produces reusable project-specific classifiers able to provide fast and accurate predictions for future test cases in that project. These classifiers are “friends” of the test suite, warning of flaky tests early on to avoid them accumulating. While Chapter 5 presented a flaky test detection approach that can offer the “best of both worlds” between rerunning- and machine learning-based flaky test detection, this study demonstrates a technique that can save time in the long run as developers introduce new test cases. Among the findings from the evaluation involving 10 Python projects are: (1) FLAKEFRIEND is able to detect existing flaky tests in a project in 20% of the time taken by exhaustive rerunning; (2) FLAKEFRIEND produces classifiers for a project that are better at detecting future flaky tests in that project than those pre-trained on the test cases of other projects (mean Matthews correlation coefficient of 0.73 versus 0.18); and (3) FLAKEFRIEND offers large savings in cumulative time cost when applied to evolving test suites, in the order of months of single-core CPU time. The study makes the following contributions:

1. **Technique (Section 6.2):** A novel technique capable of detecting and classifying existing flaky tests in a project, and of producing machine learning classifiers to provide fast and accurate predictions for future test cases.
2. **Dataset (Section 6.3):** Over several months of compute time, I collected a large dataset of test run information regarding 10,000 executions of 63,090 test cases. This is freely available in the replication package [205] to enable future testing research.
3. **Evaluation (Section 6.4):** An extensive empirical evaluation involving 10 open-source projects shows how FLAKEFRIEND addresses the drawbacks of both rerunning-based and machine learning-based flaky test detection.

Chapter 2

A Survey of Flaky Tests

The contents of this chapter is based on “O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. A survey of flaky tests. *Transactions on Software Engineering and Methodology*, 31(1):1–74, 2021”.

2.1 Introduction

The exact definition of what constitutes a flaky test varies slightly from source to source, but is generally considered to mean a test case that can be observed to both pass and fail without changes to the test case code or the code under test. Some seem to take a fairly conservative view, only considering a test case flaky if the flakiness stems from timing or concurrency issues [155]. Most sources simply define a flaky test as a test case that can be observed to produce an inconsistent outcome when only the test case code and the code under test that it exercises remains constant [105]. This particular definition is inclusive of test cases that are flaky due to external factors, such as the prior execution of other test cases, or the execution platform itself.

The most common type of flaky test related to the prior execution of other test cases are known as *order-dependent* (OD) flaky tests, one whose outcome depends on the test execution order but is otherwise deterministic [175]. Recently, OD flaky tests, and the test cases on which they depend, have been categorised by their manifestation [141], and their definition has been generalised by one study’s authors who considered how non-deterministic test cases can also be OD [92]. While it might seem initially unlikely that the test run order would change between test suite runs, several widely studied testing techniques do just that, making OD flaky tests an obstacle to their applicability and thus a genuine problem rather than just a peculiarity [15, 22, 91].

As for flaky tests related to the execution platform itself, a test case with a platform dependency may be considered flaky because it might pass on one machine but fail on another. Some may consider this to be a deterministic failure rather than a form of flakiness. However, researchers have commented that when using a cloud-based continuous integration service, where different test runs may be executed on different machines in a manner that appears non-deterministic to the user, the test outcome is effectively non-deterministic too [37]. One reason why a test case may be dependent on a particular platform is because it relies on the behaviour of a particular implementation of an underdetermined specification that leaves certain aspects of a program’s behaviour undefined [60]. For example, a test case that expects a particular iteration order for an object with no iteration order mandated by its specification may pass on one platform where the respective implementation happens to meet its expectations but fail on another [61, 140].

At least one source describes a test case as flaky if its *coverage*, the set of program elements that it executes, is also inconsistent [139]. Generalizing further, Strandberg et al. [147] defined *intermittent tests* as those with a history of passing and failing, regardless of software or hardware changes. These authors considered flaky tests to be a special case of intermittent tests that pass and fail in the absence of changes, as per the traditional definition.

Flaky tests challenge the assumption that a test failure implies a bug, constituting a leading

cause of “false alarm” test failures [155, 175] and causing problems for both developers and researchers. For developers, flaky tests erode trust in test suites and lead to time wasted debugging spurious failures [180]. A recent survey of industrial developers demonstrated the prevalence and prominence of flaky tests [37]. The results indicated that test flakiness was a frequently encountered problem, with 20% of respondents claiming to experience it monthly, 24% encountering it on a weekly basis and 15% dealing with it daily. In terms of severity, of the 91% of developers who claimed to deal with flaky tests at least a few times a year, 56% described them as a moderate problem and 23% thought that they were a serious problem.

This research area is of rapidly growing interest. Of the 76 papers in this literature survey, 63% were published between 2019 and 2021. In 2020, Zolfaghari et al. [179] provided a review on some of the main techniques with regards to the detection, repair, and root causes of flaky tests. However, to date there has been no comprehensive survey published that covers not only this core literature but also the wider periphery, such as the influence that flaky tests have in the wider software engineering field (see Section 2.4 for additional details). It is thus my position that doing so at this time of increasing attention will be most valuable, particularly to those researchers and developers wishing to familiarise themselves with the field of flaky test research.

In this survey, I examine the peer-reviewed literature directly concerning flaky tests, according to the definitions previously described. I make available a bibliography of the examined literature in a public GitHub repository [224]. Overall, this survey makes the following contributions:

1. **Evaluation (Section 2.2):** An evaluation of the growing research area of flaky tests.
2. **Causes (Section 2.3):** A comparative analysis on the causes of flaky tests.
3. **Costs (Section 2.4):** A summary of costs and impacts on testing reliability and efficiency.
4. **Detection (Section 2.5):** A comparison of tools and strategies for detecting flaky tests.
5. **Mitigation (Section 2.6):** An analysis of techniques to mitigate and repair flaky tests.
6. **Discussion (Section 2.7):** A discussion of research threads, trends, and future directions.

In Section 2.2, I present the research questions, describe the scope of the survey and explain the methodology for collecting relevant studies. In Section 2.3, I discuss the underlying mechanisms and factors I identified as the causes of flakiness, both in general and in more application-specific domains, and present a list of flakiness categories that emerge from the literature. In Section 2.4, I consider the negative impacts that flaky tests impose upon the reliability and efficiency of testing, as well as upon a range of specific testing-related activities. In Sections 2.5 and 2.6, I consider approaches for detecting, mitigating and repairing flaky tests, including insights and general techniques from the literature as well as a tour of the automated tools available. In Section 2.7, I take a high-level overview of all the sources I consulted and examine their demographics and emergent research trends before summarizing in Section 2.9. In Section 2.8, I provide an update to the survey by reviewing relevant papers published between April 2021 and April 2023.

2.2 Methodology

In this section, I present this survey’s four research questions and go on to describe its scope and give the inclusion and exclusion criteria. Following this, I describe the paper collection approach and present the results. The research questions, with their relevant sections, are as follows:

RQ1: What are the causes and associated factors of flaky tests? This explores the mechanisms behind flakiness. The answer explains the common patterns and categories, as identified by previous research, and goes on to examine more domain-specific factors. (See Section 2.3.)

RQ2: What are the costs and consequences of flaky tests? This addresses the costs of flaky tests and their severity as a problem for developers and researchers. (See Section 2.4.)

Table 2.1: The inclusion and exclusion criteria for papers to be considered in this survey.

Inclusion	Exclusion
Discusses the causes of flaky tests	Not a peer-reviewed publication
Discusses the factors associated with flaky tests	Presents the same study as another included paper
Presents a strategy for detecting or analysing flaky tests	Discusses redundant tests using the term dependent tests
Presents a strategy for repairing or mitigating flaky tests	A position paper, if the proposed work has been published
Describes an impact of flaky tests upon some technique or concept	

Table 2.2: Top five most common publication venues.

Venue	Papers
Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	14
International Conference on Software Engineering	13
International Symposium on Software Testing and Analysis	9
International Conference on Automated Software Engineering	8
International Conference on Software Testing, Verification and Validation	5

RQ3: What insights and techniques can be applied to detect flaky tests? This catalogues the techniques that have been applied to identify flaky tests, many of which have been implemented as automated tools and scientifically evaluated with real-world test suites. (See Section 2.5.)

RQ4: What insights and techniques can be applied to mitigate or repair flaky tests? This examines the range of literature presenting attempts at limiting the negative impacts of flaky tests (as I identified by answering **RQ2**) or repairing them outright. (See Section 2.6.)

2.2.1 Survey Scope

The scope of this survey is limited to peer-reviewed publications relevant to test flakiness, including those regarding order- and implementation-dependent flaky tests. I included papers that have flakiness as their main topic or are tangentially related, such as those describing a cause of flaky tests or presenting a method that is impacted by them. Other testing topics share similar terminology to test flakiness but are not directly related and I took care to exclude them. For example, I did not include works regarding redundant tests, which are sometimes described as *dependent tests* [83, 149, 150, 158]. In that context, the term is used to refer to test cases whose outcome can be inferred from the outcomes of one or more other test cases, as opposed to those having non-deterministic outcomes with respect to the test run order.

By only considering peer-reviewed sources, I excluded preprints, theses, blog posts and other “grey” literature. I also took care to exclude position papers if the proposed work has been completed and published (as in the case of Bell et al. [13, 15]) and instances where the same (or a similar) study is presented in multiple publications (such as Luo et al. [104, 105]). The inclusion and exclusion criteria are in Table 2.1. I consider a study relevant to this survey if it meets at least one of the inclusion criteria and none of the exclusion criteria.

2.2.2 Collection Approach

To collect the papers for the survey, I conducted query searches on the 15th of April 2021 using the ACM Digital Library [198], IEEE Xplore [239], Scopus [272] and SpringerLink [232]. I selected query terms that would cover both the general and order-dependent flaky test literature. The full

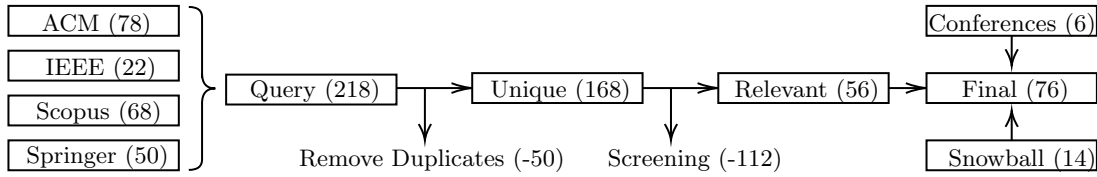


Figure 2.1: An illustration of the paper collection approach, with the number of papers at each stage.

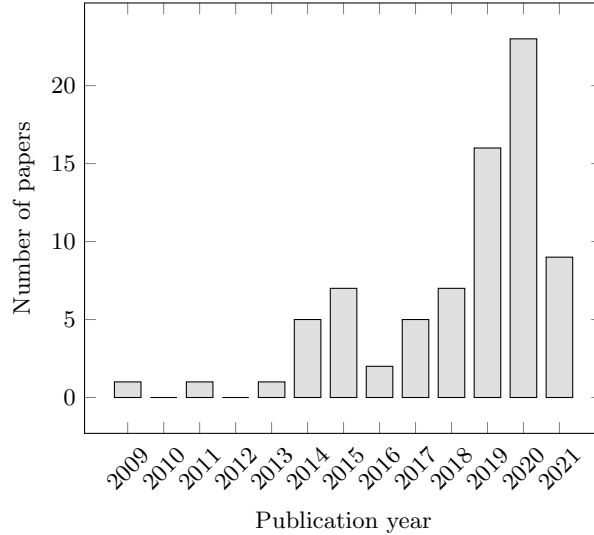


Figure 2.2: Number of papers by publication year. Searches were conducted in April 2021.

search query was (“*flaky test*” OR “*test flakiness*” OR “*intermittent test*” OR “*order dependent test*” OR “*test order dependency*”). While the subject area of the ACM Digital Library and IEEE Xplore is limited to computing/technology, Scopus and SpringerLink are general search engines including medicine, the social sciences, and other unrelated areas. For these two search engines, I made sure to restrict the results to computer science only. When evaluating papers against the inclusion and exclusion criteria, I read their titles, abstracts, introductions and evaluated their further sections if necessary. This constituted the screening process for selecting relevant papers.

It is my position that the peer-review process constitutes a sufficient quality bar and so I did not include an additional quality assessment stage in the methodology. To collect more publications beyond the query searches, I checked the accepted papers of relevant 2021 conferences (such as those in Table 2.2) that had not yet published their proceedings, and thus would not have been found as part of the query searches. In addition, I applied a technique known as *backward snowballing* [165]. This involved reading the related work sections and references of the collected papers, from which I extracted additional studies, subject to the inclusion and exclusion criteria.

2.2.3 Collection Results

Figure 2.1 illustrates the paper collection process and gives the number of papers at each stage. Following the query searches, I ended up with a total of 168 unique results (after removing duplicates). After the screening process, I selected 56 papers with respect to the inclusion and exclusion criteria as previously explained. I identified a further six relevant papers from the 2021 accepted papers of the *International Conference on Software Engineering* (ICSE) [237], the *International Conference on Software Testing, Verification and Validation* (ICST) [238], and the *International Conference on Mining Software Repositories* (MSR) [256]. Finally, I collected an additional 14 papers via backward snowballing, making for a total of 76 relevant papers.

- 2009 • First reference to flaky tests in the research literature [87].
- 2011 • First attempt at detecting order-dependent tests [113].
- 2014 • First empirical evaluation of flaky tests.
 Ten categories of flakiness defined [105].
 First empirical evaluation of order-dependent flaky tests.
 First tool for their detection, DTDETECTOR [175].
 First tool for mitigating order-dependent flaky tests, VMVM [16].
- 2015 • First tool for explicitly detecting state-polluting tests, POLDET [62].
- 2016 • First tool for detecting implementation-dependent flaky tests, NONDEX [61, 140].
- 2017 • First empirical studies of flaky tests in the context of continuous integration [71, 86, 110].
- 2018 • First tool for detecting flaky tests based on differential coverage, DEFLEAKER [16].
- 2019 • First tool for detecting both order-dependent and non order-dependent flaky tests, IDFLAKIES [90].
 First tool for automatically fixing order-dependent flaky tests, IFIXFLAKIES [141].
 First tool to apply natural language processing for the static prediction of flaky tests, FLAST [159].
- 2020 • First tool for detecting flaky tests in machine learning applications, FLASH [35].
- 2021 • First large-scale empirical study of general flaky tests in Python [59].

Figure 2.3: Timeline of research milestones in the field of flaky tests.

As shown in Figure 2.2, research interest in flaky tests appears to be increasing over time, as evidenced from the growing number of collected papers published through the years. Until 2014, research in the area was fairly limited, at which point several now heavily cited papers emerged. Namely, these were Luo et al.’s *An Empirical Study of Flaky Tests*, that introduced a widely used set of flakiness categories [105] (see Table 2.4), and Zhang et al.’s *Empirically Revisiting the Test Independence Assumption*, which presented their tool DTDETECTOR for detecting order-dependent flaky tests [175] (see Section 2.5.2). I consider these studies to be important research milestones, as shown in Figure 2.3. The trend indicates that flaky tests are an area of increasing interest, with just under 63% of all sampled studies published between 2019 and 2021.

Table 2.2 shows the top five conferences where the collected papers were published. The *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE) [233] is first, followed closely by ICSE. Both are very mature conferences for the field of computer science, with the former dating back to the year 1987 and the latter to 1975. Three of the top five conferences are in the field of software engineering, of which software testing is a subfield. This result suggests that flaky tests are prominent enough as a topic to be relevant in the more general field of software engineering, as opposed to being just a peculiarity of software testing. Together, these top five conferences account for approximately 64% of the studies examined in this survey. Thus, I recommend that interested researchers regularly check the proceedings of these conferences for new publications about flaky tests.

2.3 Causes and Associated Factors

This section reviews the studies that address the underlying causes and mechanisms that lead to flaky tests. Some of these studies, like Luo et al. [105], identified common patterns between flaky tests and categorised them by their perceived causes. Others examined more general factors, such as Shi et al. [140], that identified incorrect assumptions regarding library specifications, like the iteration order of unordered collection types, as a potential avenue for flakiness. This section answers **RQ1**, which I achieve by splitting it into three sub-research questions, each with their own associated subsections. The answers to each sub-research question are summarised at the end of their respective subsections, and their combined findings and implications, constituting the overall answer to **RQ1**, are summarised in Table 2.3. The three sub-research questions are:

RQ1.1: What are the general causes of flaky tests? This explores the different general causes of flakiness as categorised by various studies. To that end, I examine four studies that analysed flaky tests in projects of no specific type or application with the aim of establishing different categories of flakiness and identifying their respective prevalences. (See Section 2.3.1.)

RQ1.2: What are the causes of order-dependent flaky tests? Motivated by their particularly negative impacts on test suite acceleration (see Section 2.4.2), this investigates the causes and factors associated with test order dependencies. (See Section 2.3.2.)

RQ1.3: What are the application-specific causes of flaky tests? This addresses the causes of flakiness that are specific to particular types of applications or testing strategies. To answer this question, I examine studies that focused on specific types of software and identified the causes of flaky tests specific to them. (See Section 2.3.3.)

Table 2.3: Summary of the findings and implications answering **RQ1**: What are the causes and associated factors of flaky tests? Relevant to researchers (🎓) and developers (</>).

Finding	Implications	Source
🎓 Issues regarding asynchronicity and concurrency appear to be the leading causes of flaky tests.	Techniques for mitigating or repairing flaky tests ought to be able to address asynchronicity and concurrency in order to be the most impactful.	[37, 89, 105, 155]
</> Platform dependencies represented 34% of sampled bug reports indicative of flaky tests.	Developers should ensure that test outcomes are consistent across the platforms targeted by their software, especially when using cloud-based continuous integration.	[37, 155]
🎓 Order-dependent tests were found to constitute up to 16% of flaky test bug reports and 9% of previous flaky test repairs.	Test order dependency appears to be a prominent category with specific negative impacts (see Section 2.4.2) and thus deserving of specific attention.	[37, 105, 155]
🎓 The majority of order-dependent tests are victims , passing in isolation but failing after the execution of certain polluter tests.	Techniques to eliminate state-pollution during the execution of tests would indirectly repair most order-dependent tests . Developers ought to be especially careful that they do not introduce test cases that may induce a failure in other tests in the test suite.	[59, 141]
🎓 The dependency of 76% of order-dependent tests was related to only one other test .	Techniques for detecting order-dependent tests should consider that complex networks of dependencies are an unlikely occurrence .	[175]
🎓 Shared access to in-memory resources via static fields was the facilitator of 61% of order-dependent tests in the Java programming language.	Approaches for dealing with test order dependency should consider that a significant minority of cases cannot be identified by analyzing program state alone .	[175]
🎓 Insufficient waiting for elements to be rendered in user interface testing appears to be a strong factor associated with flakiness in this domain.	Due to their similarities as timing-based dependencies , insights into mitigating asynchronous wait flakiness could be useful in this context.	[49, 132, 136]
🎓 Algorithmic non-determinism accounts for 60% of flaky tests in machine learning projects .	Strategies for dealing with flakiness applied to more general projects, where asynchronous waiting and concurrency are the leading causes of flakiness, may not be so applicable to machine learning projects.	[35]
🎓 The distribution of causes of flaky tests within Android applications appears to roughly correspond to that of more general projects created in languages like Java.	Insights gained from studying flaky tests in general are likely to be relevant to Android projects.	[152]

2.3.1 General Causes

Studies have examined and categorised the causes of flaky tests in general software projects, that is, not constrained to a particular platform or purpose. To that end, objects of study such as historical commit data [105], bug reports [155], developers’ insights from flaky tests that they have previously repaired [37], pull requests [89] and execution traces [59] have been analysed, resulting in a commonly used set of flakiness categories. Two of these studies consider subjects from the same source, namely the Apache Software Foundation [280], yet interestingly present different findings [105, 155]. The union of the categories used in each of these sources is presented and described in Table 2.4. Inspired by previous work [161], I also present each category as a member of one of three families in order to illustrate their relatedness. The *intra-test* family describes flakiness wholly internal to the test, in other words, stemming from issues isolated to the execution of the test code itself, or the direct code under test. The *inter-test* family contains tests that are flaky with respect to the execution of other tests, for example, those with test order dependencies. The flakiness of tests in the *external* family stems from factors outside of the test’s control, such as the time taken to receive a response from a web sever. Table 2.5 presents the prevalence of these categories as determined by each source, along with their type of subject.

Table 2.4: Categories used to classify flaky tests in various studies.

Family	Category	Description	Source
Intra-test	Concurrency	Test that invokes multiple threads interacting in an unsafe or unanticipated manner. Flakiness is caused by, for example, race conditions resulting from implicit assumptions about the ordering of execution, leading to deadlocks in certain test runs.	[37, 59, 89, 105, 155]
	Randomness	Test uses the result of a random data generator. If the test does not account for all possible cases, then the test may fail intermittently, e.g., only when the result of a random number generator is zero.	[37, 59, 89, 105]
	Floating Point	Test uses the result of a floating point operation. Floating point operations can suffer from a variety of discrepancies and inaccuracies such as precision over and under flows, non-associative addition, etc., which if not properly accounted for can result in inconsistent test outcomes, e.g., by comparing the result of a floating point operation to an exact real value in an assertion.	[37, 89, 105]
	Unordered Collection	Test assumes a particular iteration order for an unordered collection type object. Since no particular order is specified for such objects, tests that assume they will iterate in some fixed order will likely be flaky due to a variety of reasons, e.g., implementation of the collection class.	[59, 89, 105]
	Too Restrictive Range	Test where some of the valid output range falls outside of what is accepted in its assertions. This test is flaky since it does not account for corner cases and thus it may intermittently fail when they arise.	[37, 59]
	Test Case Timeout	Test specified with an upper limit on their execution time. Since it is usually not possible to precisely estimate how long a test will take to run, by specifying an upper time limit a developer may run the risk of creating a flaky test, since it could fail on certain executions that are slower than anticipated.	[37, 59]
	Inter-test	Test Order Dependency	Test that depends on some shared value or resource that is modified by another test which impacts its outcome. In the case where the test run order is changed, these flaky tests may produce inconsistent outcomes since the dependencies upon tests previously executed beforehand are broken.

	Resource Leak	Test that improperly handles some external resource, e.g., failing to release allocated memory. Improperly handled resources may cause flakiness in subsequently executed test cases which attempt to reuse that resource.	[37, 59, 89, 105, 155]
	Test Suite Timeout	Test is part of a test suite with a limited execution time. Intermittently fails because it happens to be running once the test suite hits an upper time limit.	[37]
External	Asynchronous Wait	Test makes an asynchronous call and does not explicitly wait for it to finish before evaluating assertions, typically using a fixed time delay instead. Results may be inconsistent in executions where the asynchronous call takes longer than the specified time to finish, leading to the flakiness.	[37, 59, 89, 105, 155]
	I/O	Test that is flaky due to its handling of input and output operations. For example, a test that fails when a disk has no free space or becomes full during file writing.	[37, 59, 89, 105]
	Network	Test that depends on the availability of a network connection, e.g., by querying a web server. In the case where the network is unavailable or the required resource is too busy, the test may become flaky.	[37, 59, 89, 105]
	Time	Test relies on local system time and it may be flaky due to discrepancies in precision and timezone, e.g., failing when midnight changes in the UTC timezone.	[37, 59, 89, 105]
	Platform Dependency	Test depends on some particular functionality of a specific operating system, library version, hardware vendor, etc.. While such tests may produce a consistent outcome on a given platform, they are still considered flaky, particularly with the rise of cloud-based continuous integration services, where different test runs may be executed upon different physical machines in a manner that appears non-deterministic to the user.	[37, 59, 155]

Commits

As part of an empirical evaluation, Luo et al. [105] sampled historical commit data from 51 projects of varying size and language (mostly Java) from the Apache Software Foundation. They identified 201 individual commits that were evidence of a flaky test being repaired by a developer and studied these to determine the most common causes of test flakiness. Specifically, they categorised the type of flakiness that each commit repaired under 11 causes, including a miscellaneous or “hard to classify” category. They categorised 37% of commits under the *asynchronous wait* category, meaning a developer repaired flaky tests that were caused by not properly waiting upon asynchronous calls. Such tests typically employed a fixed time delay following the invocation of some asynchronous operation instead of explicitly waiting for it to complete before evaluating assertions, potentially leading to an erroneous outcome in the executions where the time delay was insufficient. Examples include waiting for a response from a web server or waiting for a thread to finish. Figure 2.4 shows a concrete example of a flaky test of the *asynchronous wait* category taken from the Home Assistant project [234]. They categorised 16% of their commits under the *concurrency* category, pertaining to flaky tests whose non-determinism was due to undesirable interactions between multiple threads, including issues such as data races and deadlocks. The main difference between this category and *asynchronous wait* is that the latter refers to synchronisation issues explicitly concerning external or remote resources. They categorised a further 9% under the *test order dependency* category, in other words, a developer fixed order-dependent tests, and 5% under *resource leak*. They distributed the remainder among the other eight categories, which included causes related to networking, time, floating point operations, assumptions regarding the iteration order of unordered collection types [140] and input/output operations. The authors suggested that techniques for detecting and fixing flaky tests ought to focus on those related to asynchronicity, concurrency and test order dependency, the three most commonly observed categories.

```

1  async def test_get_image(hass, hass_ws_client, caplog):
2      """Test get image via WS command."""
3      await async_setup_component(
4          hass, "media_player", {"media_player": {"platform": "demo"}}
5      )
6      await hass.async_block_till_done()
7
8      client = await hass_ws_client(hass)
9
10     with patch(
11         "homeassistant.components.media_player.MediaPlayerEntity."
12         "async_get_media_image",
13         return_value=(b"image", "image/jpeg"),
14     ):
15         await client.send_json(
16             {
17                 "id": 5,
18                 "type": "media_player_thumbnail",
19                 "entity_id": "media_player.bedroom",
20             }
21         )
22         msg = await client.receive_json()
23         assert msg["id"] == 5
24         assert msg["type"] == TYPE_RESULT
25         assert msg["success"]
26         assert msg["result"]["content_type"] == "image/jpeg"
27         assert msg["result"]["content"] == base64.b64encode(b"image").decode("utf-8")
28         assert "media_player_thumbnail is deprecated" in caplog.text

```

Figure 2.4: A flaky test from the `home-assistant` project [234]. This test case was flaky until the addition of line 6 [222] (highlighted), which blocks the calling thread until all pending work to setup the components under test has been completed. Previously, the test case might fail if this had not been completed before the assertions starting on line 23 were evaluated.

Bug Reports

Vahabzadeh et al. [155] categorised the root causes of test bugs, that is, bugs in the code of tests as opposed to the code under test, which they mined from the bug repository of the Apache Software Foundation. In total, they systematically categorised 443 test bugs. They considered five categories: *semantic bugs*, *environment*, *resource handling*, *flaky tests*, and *obsolete tests*. Based upon their descriptions, I would consider the *environment* and *resource* categories, along with *flaky tests* of course, to fall under the more inclusive definition of flakiness given in Section 2.1. From this perspective, they categorised 51% of test bugs to be flaky tests, 34% of which under the *environment* category, which was sub-categorised into tests that were inconsistent across operation systems and those that were inconsistent across third party library or Java Development Kit versions and vendors. For the purposes of Table 2.5, I consider these as part of the *platform dependency* category. The *resource* category was sub-categorised into *test order dependency*, accounting for 16% of flaky tests, and *resource leak*, accounting for 8%. Finally, the *flaky tests* category consisted of the *asynchronous wait*, *race condition* and *concurrency bugs* subcategories. Considering the latter two as part of the more general *concurrency* category of Table 2.5, they categorised 18% of flaky tests under *asynchronous wait* and 19% under *concurrency*. The results of this study are different from the previous work of Luo et al. [105], with *asynchronous wait* being only the third most common category, for example. This may be surprising given that they both examined subjects from the same source (i.e., the Apache Software Foundation). However, this study takes bug reports as objects of study, which developers may not have addressed, whereas Luo et al. explicitly analysed commits that *fixed* flaky tests. Therefore, this difference in the results reported by Luo et al. and

Table 2.5: The prevalences of the causes of flaky tests as categorised by various studies. Dashes indicate that the study did not consider that category. Percentages are rounded and do not sum to 100% since not all flaky tests could be categorised by the authors of the referenced source.

Source	Subjects	Concurrency	Randomness	Floating Point	Unordered Coll.	Too Res. Range	Case Timeout	Test Order Dep.	Resource Leak	Suite Timeout	Async. Wait	I/O	Network	Time	Platform Dep.
[105]	Commits	16%	2%	2%	1%	-	-	9%	5%	-	37%	2%	5%	3%	-
[155]	Bug reports	19%	-	-	-	-	-	16%	8%	-	18%	-	-	-	34%
[37]	Fixed flaky tests	26%	1%	3%	0%	17%	8%	9%	6%	2%	22%	0%	0%	2%	4%
[89]	Pull requests	8%	5%	2%	0%	-	-	0%	5%	-	78%	5%	14%	4%	-
[59]	Execution traces	3%	37%	-	1%	0%	1%	-	2%	-	3%	7%	42%	4%	0%

Vahabzadeh et al. suggests that there may be certain categories of test flakiness that developers are more likely or more able to repair.

Developer Survey

Eck et al. [37] asked 21 software developers from Mozilla to classify 200 flaky tests that they had previously fixed, using the categories of Luo et al. [105] as a starting point, but being allowed to create new ones if appropriate. After reviewing the developers' responses, they identified four new emergent categories. The first was *test case timeout*, which refers to the circumstance in which a test is flaky due to sometimes taking longer than some specified upper limit for single test case executions. Another related category was *test suite timeout*, in which a test times out non-deterministically as before but due to a time limit on the whole test suite execution. Another was *platform dependency*, where a test unexpectedly fails when executed on different platforms, such as across Linux kernel versions, even if the failure is consistent. The final new category was *too restrictive range*, in which some valid output values lie outside of assertion ranges, making the test fail in these corner cases. Their results showed that the top three categories observed by the developers were *concurrency*, *asynchronous wait* and *too restrictive range*, accounting for 26%, 22% and 17% of cases, respectively. The *test order dependency* category, one of the top three categories in the previous work of Luo et al., came in fourth, responsible for 9% of cases. Since these insights were also derived from previously *fixed* flaky tests, one could argue that their results are more directly comparable to those of Luo et al. than those of Vahabzadeh et al. [155]. This statement is supported by the fact that the most commonly identified causes of flakiness in this study are similar to that of Luo et al. [105], albeit with *test order dependency* not quite making the top three.

Pull Requests

Lam et al. [89] categorised the type of flakiness repaired in 134 pull requests regarding flaky tests in six subject projects that were internal to the Microsoft corporation. These projects used Microsoft's distributed build system CLOUDBUILD, which additionally contains FLAKES, a flaky test management system. By re-executing failed tests, FLAKES identifies those that are flaky and generates a corresponding bug report. By examining the bug reports that had been addressed by developers via a pull request, they were able to categorise the respective fixes, using the categories from previous work [105, 118]. They identified *asynchronous wait* as a leading category, though with a significantly higher prominence of 78% of pull requests, in comparison to 37% of commits as found previously by Luo et al. [105]. Furthermore, they found no cases of the *test order dependency* category, which had previously been identified as being common. One possible reason for this is

that FLAKES only re-executes tests in their original order and does no reordering, therefore making it unlikely to identify, and generate bug reports for, any order-dependent tests.

Execution Traces

Gruber et al. [59] performed an empirical analysis of flaky tests in the Python programming language. To collect subjects, they automatically scanned the Python Package Index [265] and selected projects with a test suite that could be executed by PyTest, a popular Python test runner [266] on which their experimental infrastructure depends. This resulted in 22,352 subject projects containing a total of 876,186 test cases. They executed each test suite 200 times in its original order and 200 times in a randomised order to identify flaky tests including order-dependent tests. Introducing the concept of *infrastructure flakiness*, Gruber et al. split each of these 200 runs into 10 iterations of 20 runs, each executed on the same machine in an uninterrupted sequence. They considered a flaky test to be due to infrastructure flakiness if it was flaky between iterations but not within an iteration — for example, a test case that fails in every run of one iteration but consistently passes every run of every other iteration. They reasoned that such instances were due to non-determinism in the testing infrastructure. For a sample of 100 flaky tests that were neither order-dependent nor due to infrastructure non-determinism, the authors classified their causes based on keywords in the execution traces of the flaky test failures. For instance, they considered keywords such as *thread* and *threading* to be indicative of flaky tests of the *concurrency* category. In total, they identified 7,571 flaky tests among 1006 projects of their subject set. Of these, they found 28% to be due to infrastructure flakiness, 59% due to test order dependencies and 13% due to other causes. Of the 100 automatically classified flaky tests (randomly sampled from the latter 13%), they found the most common category to be *network*, accounting for 42%. This was followed by *randomness* at 37%. At odds with the general trend of previous studies [37, 89, 105, 155], *asynchronous wait* and *concurrency* were minority categories, responsible for 3% of the categorised flaky tests each. The authors put this down to use case differences between the Python and Java programming languages, since these previous studies focused predominantly on subjects of the latter.

Conclusion for RQ1.1: What are the general causes of flaky tests? The overarching picture painted by studies examining commit data, bug reports, developer survey responses, and pull requests suggests that timing dependencies related to asynchronous calls is the leading cause of test flakiness [37, 89, 105, 155]. One study that categorised pull requests of automatically generated bug reports of flaky tests found the prevalence of this category to be as high as 78%. Therefore, techniques for mitigating or automatically identifying such cases would likely address the most significant share of flaky tests. However, another study examining execution traces specifically from projects implemented in the Python programming language [59] found this to be a minority category. Other emergent leading causes include concurrency related issues, other than asynchronicity and platform dependency. The latter of these, accounting for up to 34% of bug reports indicative of flaky tests in one study [155], is particularly relevant in the age of cloud-based continuous integration, where test runs may be scheduled across heterogeneous machines in a seemingly non-deterministic manner depending on availability [37]. Another prominent cause of flaky tests are test order dependencies (i.e., order-dependent tests), representing 9% of flaky test repairs in two studies [37, 105].

2.3.2 Test Order Dependencies

Multiple sources have specifically considered the causes of, and the factors associated with, the *test order dependency* category of flaky tests. Some of these have discussed the difficulties associated with implementing and executing procedures between test case runs for resetting the program state, specifically setup and teardown methods [14, 67]. Others have examined the impact of global variables, such as *static fields* in the Java programming language, as a vector for conveying

```

1  @pytest.mark.parametrize(
2      "config_dir", ["../hydra/test_utils/configs"],
3  )
4  @pytest.mark.parametrize(
5      "config_file, overrides, expected",
6      [
7          (None, [], {}),
8          (None, ["foo=bar"], {"foo": "bar"}),
9          ("compose.yaml", [], {"foo": 10, "bar": 100}),
10         ("compose.yaml", ["group1=file2"], {"foo": 20, "bar": 100}),
11     ],
12 )
13 class TestCompose:
14     def test_compose_decorator(
15         self, hydra_global_context, config_dir, config_file, overrides, expected
16     ):
17         with hydra_global_context(config_dir=config_dir):
18             ret = hydra.experimental.compose(config_file, overrides)
19             assert ret == expected
20
21     def test_strict_failure_global_strict(
22         self, hydra_global_context, config_dir, config_file, overrides, expected,
23     ):
24         overrides.append("fooooooooo=bar")
25         with hydra_global_context(config_dir=config_dir, strict=True):
26             with pytest.raises(KeyError):
27                 hydra.experimental.compose(config_file, overrides)

```

Figure 2.5: An example of an order-dependent flaky test from the `hydra` project [220]. The test case `test_strict_failure_global_strict` appends a string to the list passed as the `overrides` argument on line 24, which is part of a class-wide test parametrisation given on line 4. The test case `test_compose_decorator` expects `overrides` to be of its initial value and would fail if executed after `test_strict_failure_global_strict`, as may other tests in `TestCompose` [236] that unexpectedly receive the modified version of `overrides`.

information between test cases and thus establishing order-dependent tests [14, 15, 113, 175]. One study posited that failing to control all the inputs that may later impact assertions leaves tests open to establishing dependencies on one another [74]. An example of an order-dependent test, and the test that can cause it to fail, is given in Figure 2.5, taken from Facebook’s Hydra project for the configuration of Python applications [220].

Classification

Shi et al. [141] introduced the terminology of *victim* and *brittle* for describing order-dependent tests. A victim test passes in isolation but fails when executed after certain other tests. The tests that cause a victim to fail are known as the victim’s *polluters*. Conversely, a brittle test fails in isolation and requires the prior execution of other tests to pass, known as *state-setters*. Shi et al. performed an evaluation using 13 modules of open-source Java projects containing 6,744 tests in total. By executing the respective test suites in randomised orders, they identified 110 order-dependent tests. Of these, 100 were victims and 10 were brittles.

As part of their empirical evaluation of flaky tests in Python projects, Gruber et al. [59] followed a similar procedure upon 22,352 Python projects from the Python Package Index [265]. They identified 4,461 order-dependent tests, 3,168 of which were victims and 738 of which were brittles. The authors were unable to categorise the remaining 555 due to limitations in their testing framework. Overall, the combination of these two studies indicates that the vast majority of order-dependent tests would pass in isolation but may fail due to the action of other tests.

Setup and Teardown

Haidry et al. [67] reasoned that test order dependencies are a result of interactions between components of the software under test. Giving an example of a hypothetical piece of software for running an automated teller machine (ATM) machine, they explained that, since the machine requires the user to enter their PIN before inputting the desired amount of cash, the test case that covers the PIN component must be executed before the test case that covers the cash withdrawal component to ensure that the system is in the required state. A developer could write each of these test cases to be independent of one another, but that would require some amount of additional setup for each of them, resulting in a less efficient test suite. Therefore, test order dependencies may be a consequence of developers prioritizing efficiency and convenience over test case independence.

Having studied the prevalence of per-test process isolation as a mitigation for test order dependencies, which involves executing each test in its own process to prevent side effects, Bell et al. [14] suggested that the burden of writing setup and teardown methods could be a factor in the existence of test order dependencies. Setup methods, which ensure that the state of the program is as expected before a test is executed, and teardown methods, which reset the state of the program to as it was before the test run, are a means of avoiding side effects between tests, meaning isolation should not be necessary. Bell et al. remarked, however, that these methods may be difficult to implement correctly, perhaps explaining their observation that 41% of 591 sampled open-source projects used isolated test runs as a catch-all attempt to eliminate side effects and thus test order dependencies.

Static Fields

Muşlu et al. [113] remarked how the implicit assumptions of developers regarding the way in which their code is executed, such as the ordering of method calls and data dependencies, leads to a class of bugs which, when testers are unaware of such assumptions, are likely to be overlooked. They explained how this can manifest itself as test cases that depend on the execution of other tests in a test suite and behave differently when executed in isolation. Such tests violate the *test independence assumption* that all tests are isolated entities, something that they stated is rarely made explicit, thus exacerbating the problem. The authors went on to describe and evaluate a technique for manifesting these implicit dependencies and found the improper use of Java static fields to be a potential vector.

Of the 96 instances of order-dependent Java tests identified by keyword searches in various issue tracking systems, Zhang et al. [175] found a majority of 73 requiring only one other test to manifest the test order dependency. In other words, 76% of their sample were observed to produce a different outcome if executed after only one other test in isolation from the rest of the test suite, compared to the outcome when running the test suite as normal. They found that 61% of order-dependent tests were facilitated by a static field, for example, one test modifies some object pointed to by a static field somewhere which is later used by a subsequent test, with the remainder caused by side effects within external resources such as files, databases or some other unknown cause. This finding roughly concurs with that of Luo et al. [105], who found that 47% of order-dependent tests were caused by dependencies on external resources.

When introducing their test virtualisation approach, Bell et al. [14] described static fields as potential points of “leakage” between test runs, becoming possible avenues for tests to share information or resources and thus become dependent upon one another. Through the evaluation of their order-dependent test detection approach in a later study, Bell et al. [15] discovered that the test order dependencies identified by their tool were facilitated by a very small number of static fields. In other words, many order-dependent tests were caused by shared access to the same static fields in a Java program. Further examination revealed that most of these were within internal Java runtime code, as opposed to application or test code, and all of them were of a primitive data type.

Table 2.6: Leading causes of flakiness in specific domains.

Source	Domain	Leading Cause
[48, 49, 132, 136]	User interface testing	Asynchronous wait
[35, 116]	Machine learning	Randomness
[152]	Mobile applications	Concurrency

Brittle Assertions

Huo et al. [74] reasoned that *brittle assertions*, or assertion statements that check values derived from inputs that the test case does not control, can give rise to order-dependent tests. Tests containing brittle assertions generally make an implicit assumption that the program state that constitutes the uncontrolled inputs will always be of some particular value, often their default value. This assumption can facilitate a test order dependency if another test modifies this program state before the evaluation of the brittle assertion. The authors gave an example of a Java test class containing a test method that assumes various instance fields of an object under test are set to their default values. In the case where a previously executed test method modifies the values of these fields, the former test becomes order-dependent, stemming from the uncontrolled, assumed default, field values.

Conclusion for RQ1.2: What are the causes of order-dependent flaky tests? Studies have attributed test order dependencies to the difficulty of implementing, and the added computational cost of executing, setup and teardown procedures between test case runs [14, 67]. In terms of how they are facilitated, one source found 61% of order-dependent tests to have been caused by global variables, specifically static fields, as opposed to external factors such as files, and that the majority, 76%, appeared to depend on only one other test [175]. Furthermore, another study found that most test order dependencies are conveyed via a relatively small number of static fields, mostly found within internal runtime code as opposed to application or test code [15]. An additional cause that has been associated with order-dependent tests is related to the assumption that certain parts of the program state will always be of some particular value. This gives rise to tests that do not fully control all the inputs evaluated within their assertions, which may go on to become order-dependent if other tests modify such inputs [74].

2.3.3 Application-Specific Causes

As well as the causes of flakiness in general projects, studies have examined the factors associated with flaky tests specific to particular applications or domains. There exists a line of work explicitly concerned with flakiness within the test suites exercising user interfaces [48, 49, 132, 136]. In addition, given the rise in popularity of machine learning techniques, studies have begun to consider the particularities of flaky tests within such projects [35, 116]. Furthermore, following a similar methodology to Luo et al. [105], one source categorised flaky tests specific to Android projects [152].

User Interface Testing

Gao et al. [48] discussed the causes of flakiness in GUI regression testing. The process of GUI regression testing requires the test harness to take measurements about particular GUI state properties, such as the positions of various elements, which are then compared with measurements taken of a later version to identify unexpected discrepancies. During the execution of test cases, GUI test oracles evaluate an extensive range of properties, including those that may be tightly coupled with the hardware on which they are executed, such as screen resolution, meaning test outcomes can be inconsistent across machines. Furthermore, since it is difficult to predict the amount of time required to render a GUI, or due to instability within the GUI state itself, the

point in time at which to take these measurements may be unclear. In other words, if measurements are taken too early, before the GUI has been fully drawn, then the values of these properties may be inaccurate. All these factors contribute to the potential for these properties to become highly unstable, something which has been quantified in terms of entropy [49].

A later study by Presler-Marshall et al. [132] evaluated the impact of various environments and configurations of SELENIUM [275], a framework for automating UI tests in web apps, upon test flakiness. Specifically, they analysed the effects of varying five properties of the execution platform. The first was the waiting method, used by SELENIUM to wait for the various UI elements to appear before evaluating test oracles. As previously explained, tests may be flaky if they are too eager to evaluate assertions before the web browser has fully rendered the page. The second was the web driver, SELENIUM's interface for controlling a web browser to execute tests. In the context of this study, the term web driver was used interchangeably with web browser, since they have a one-to-one correspondence with respect to their evaluation. The web drivers they evaluated were CHROME [227], FIREFOX [221], PHANTOMJS [261] and HTMLUNIT [235], the latter two being *headless*, meaning that they do not render a graphical display and are used primarily for testing and development purposes. The third factor was the amount of memory available to the testing process, doubling from 2GB up to 32GB. The fourth was the processor, which they varied between a notebook CPU, supposed to represent what a software engineering student might use, and a much more powerful workstation processor. The final factor was the operating system, specifically Windows 10 or Ubuntu 17.10. The object of their analysis was an on-going software engineering student project containing nearly 500 SELENIUM tests. Under five configurations of the five described properties, they counted the number of test failures after 30 test suite executions, knowing that tests ought to pass and so a failure would be indicative of flakiness. The authors found that Java's `Thread.sleep` [277] with a fixed time delay as a waiting method yielded the fewest test failures of all the methods evaluated, compared with not waiting at all or using more dynamic, conditional waiting methods. They also found that a faster CPU resulted in substantially fewer flaky tests, presumably since it would render web pages fast enough to ensure that the waiting method would have less of an impact.

Romano et al. [136] performed an empirical evaluation examining, among other things, the causes of flakiness in user interface testing. As subjects, they took both web and Android applications, and, following a methodology inspired by Luo et al. [105], manually inspected 3,516 commits from 7,037 GitHub repositories. Eventually, they narrowed down their objects of analysis to 235 distinct flaky tests and categorised them by their root causes into four broad categories, each broken down into subcategories. Their first broad category, *asynchronous wait*, contained 45% of the 235 flaky tests and consisted of three subcategories. The first subcategory was *network resource loading* and describes flaky tests attempting to manipulate remote data or resources that, depending on network conditions, are not fully loaded (or have failed to load at all). The second was *resource rendering* and refers to the case where a flaky test attempts to perform an action upon a component of the user interface before it is fully drawn. The third was *animation timing*, containing flaky tests that rely on animations in the user interface and are thus sensitive to timing differences across different running environments. Their second broad category was *environment*, consisting of two subcategories, and containing 19% of their dataset. The first subcategory was *platform issue*, regarding issues with one particular platform that may cause flakiness, and the second was *layout difference*, pertaining to differences in layout engines in different internet browsers. The third broad category was *test runner API issue*, accounting for 17% of the flaky tests. This was broken down into *DOM selector issue*, problems interacting with the Document Object Model (DOM) due to differences in browser implementation, and *incorrect test runner interaction*, incorrect behaviour within APIs provided by the test runner. Finally, the last broad category was *test script logic issue* and contained 19% of the flaky tests. This was broken down into the familiar *unordered collections*, *time*, *test order dependency*, and *randomness*, and contained the additional subcategory *incorrect resource load order*, describing flaky tests that load resources after the calls that load the tests, causing the tested resources to be unavailable at test run time.

Machine Learning

Nejadgholi et al. [116] studied *oracle approximations* in the test suites of deep learning libraries. An oracle approximation in this context refers to the range of acceptable output from some deep learning-based software under test, which are naturally probabilistic in nature, as estimated by a developer. These ranges are expressed as assertion statements within test cases which, due to their probabilistic nature, often take the form of a range of acceptable numerical values or some approximate equality operator with a defined tolerance. This is in contrast to, for example, a simple equality test with a pre-defined value that might be applicable when testing a deterministic algorithm but may be inappropriate in this context. They proposed a set of assertion patterns that express oracle approximations and identified instances of them across four popular deep learning libraries implemented in the Python programming language. Across these projects, between 5% and 25% of all assertions were deemed to be oracle approximations. Since an oracle approximation may be overly conservative in certain circumstances, this may lead to tests failing inconsistently for corner cases and thus being flaky tests as part of the *too restrictive range* category [37].

Dutta et al. [35] argued that machine learning and probabilistic applications suffer especially from test flakiness as a result of non-systematic testing specific to their domain. They examined 75 previously fixed flaky tests from the commit and bug report data of four popular open-source machine learning projects written in Python and categorised their causes and fixes. They devised their own categories to be more specific regarding the origin of the flakiness in their particular subject set, explaining that under the categories of Luo et al. [105], they would have mostly been categorised under *randomness*, which would be too general for this more specific study. They categorised the majority of flaky tests, 60% of those sampled, as being caused by *algorithmic non-determinism*, that is, non-determinism inherent in the probabilistic algorithms under test common in machine learning projects, such as Bayesian inference and Markov Chain Monte Carlo among others. For instance, they described the situation in which a developer wishes to test a statistical model, selecting a small dataset upon which to train it. If the developer does not account for the non-zero chance that the training algorithm does not converge after some number of iterations, then any assertions regarding the parameters of the trained model may fail in some cases. They associated five of their sampled tests with incorrect handling of floating point computations (e.g., rounding errors and the mishandling of corner cases such as NaN). A further four were categorised under incorrect API usage. Unsynchronised seeds — in which the program inconsistently set the seeds of different random number generators within different libraries — were blamed for two flaky tests. The remainder were split between concurrency or hardware related causes, or were labelled by the authors as *other* or *unknown*.

Mobile Applications

Thorve et al. [152] claimed that Android apps have a particular set of characteristics that may make them specifically vulnerable to test flakiness. These include *platform fragmentation* — the vast number of versions and variants of the Android operating system available — and the *diversity of interactions* — the functionality within Android apps which typically involves many third-party libraries, networks and hardware with highly variable specifications (e.g., screen size and processing power). With a methodology inspired by Luo et al. [105], they searched through the historical commit data of Android projects on GitHub for particular keywords associated with flaky tests. In total, they identified 77 commits across 29 Android projects that appeared to fix flaky tests. They categorised 36% as related to concurrency. Unlike Luo et al. [105], they considered the *asynchronous wait* category as part of the *concurrency* category. They categorised 22% as based upon assumptions or dependencies regarding specific hardware, the Android version or the version of particular third-party libraries. A further 12% they categorised as being attributable to erroneous program logic, specifically regarding corner-case program states. The remaining flaky tests were categorised as being caused by either network or user interface related non-determinism, or were too difficult to classify.

Fazzini et al. [42] highlighted the significant number of interactions between mobile apps and

their software environment as a cause of test flakiness. Specifically, they referred to external calls to application frameworks that collect data from the network, location services, camera, and other sensors. They explained that a common strategy to mitigate such flakiness is to create test mocks for these sorts of external calls, providing tests with fictitious but deterministic data [145].

Conclusion for RQ1.3: What are the application-specific causes of flaky tests? In the context of user interface testing, an insufficient waiting mechanism to allow the interface to be fully rendered and stable before executing tests appears to be strongly associated with flakiness in this domain, in a manner comparable to that of flaky tests related to asynchronous waiting [48, 132, 136]. For machine learning projects, overly conservative approximations regarding the valid output of probabilistic algorithms and the high degree of algorithmic non-determinism in general appears to be a major cause of flaky tests [35, 116]. The distribution of causes of flaky tests within Android applications appears to roughly correspond to what has been observed in more general efforts to categorise flakiness [37, 89, 105, 155], with concurrency and dependencies upon the highly variable execution platform identified as major causes [152].

2.4 Costs and Consequences

This section presents evidence of the negative consequences of test flakiness to illustrate its prominence as a problem. Following the same methodology as outlined in Section 2.2, the goal of this section is to answer **RQ2** by posing three sub-research questions, with findings summarised in Table 2.7. These three questions aim to capture the variety of discussions and analyses in the literature regarding the impacts and costs of flaky tests, and are as follows:

RQ2.1: What are the consequences of flaky tests upon the reliability of testing? This provides insights into the extent to which test flakiness detracts from the value of testing by casting doubt on the meaning of a test case’s outcome. Consulting studies that analysed issue trackers and bug reports, my answer examines how flaky tests manifest themselves with respect to incorrectly indicating the presence of a bug or allowing one to go unnoticed. I go on to consider the impact that ignoring flaky tests can have on the volume of crashes experienced by users, demonstrating that, while unreliable, flaky tests may still provide some testing value and should not always be immediately deleted by developers. (See Section 2.4.1)

RQ2.2: What are the consequences of flaky tests upon the efficiency of testing? This studies how flaky tests could impose costs on the time taken to receive useful feedback from test suite executions. By reviewing several empirical studies, my answer specifically examines how a particular category of flaky test poses a threat to the validity of a variety of techniques designed to accelerate the testing process. (See Section 2.4.2.)

RQ2.3: What other costs does test flakiness impose? This is a catch-all for any relevant sources not consulted when answering the previous questions. My answer aims to identify all other testing-related areas that are negatively affected by flaky tests. (See Section 2.4.3.)

Table 2.7: Summary of the findings and implications answering **RQ2**: What are the costs and consequences of flaky tests? Relevant to researchers (☎) and developers (</>).

Finding	Implications	Source
</> Of 107 tests identified as flaky by repeatedly executing whole test suites, only the flakiness of 50 could be reproduced by repeatedly executing them in isolation .	Flaky tests seem more consistent when run in isolation, meaning attempts to reproduce their flakiness this way might be challenging, but reproducing their failure for debugging purposes may still be effective .	[92]

</> Out of 96 sampled order-dependent tests, 94 reportedly caused a test failure in the absence of a bug, i.e., a false alarm , the remainder caused a bug to go unnoticed. Out of a sample of 443 test bug reports, 97% led to a false alarm, and of these, 53% were flaky tests.	The dominant consequence of flaky tests appears to be false alarms and flaky tests also appear to be their leading cause. This indicates that flaky tests may be a considerable waste of a developer's time since they may attempt to go looking for a bug that does not exist.	[155, 175]
</> Production builds of the Firefox web browser with one or more flaky test failures were associated with a median of 234 crash reports . Builds with one or more test failures in general were associated with a median of 291.	Despite being less reliable, flaky test failures should not be ignored , in the same way that any test failure should not be ignored.	[37]
</> Of the test execution logs of over 75 million builds on Travis, 47% of previously failing, manually restarted builds subsequently passed, indicating that they were affected by flakiness. Projects with restarted builds experienced a slow down of their pull request merging process of 11 times compared to projects without.	Flaky tests can threaten the efficiency of continuous integration by intermittently failing builds and requiring manual intervention, hindering the process of merging changes and stalling a project's development.	[33]
🎓 At Google , of the 115,160 test targets that had previously passed and failed at least once, 41% were flaky. Of the 3,871 distinct builds sampled from Microsoft's distributed build system, 26% failed due to flakiness.	Test flakiness has been reported within all sectors of the software engineering field , from independent open-source software projects to the proprietary products of some of the world's largest companies.	[88, 110]
</> Over a 30-day period of test execution data, 0.02% of test executions resulted in flaky failures, though had these not been automatically identified by the testing infrastructure, 5.7% of all failed builds would have been due to these flaky tests .	A relatively small number of flaky tests in a test suite can have a significant impact on the number of subsequently failed builds as part of a continuous integration system.	[89]
🎓 </> Across the developer-written test suites of 11 Java modules, 23% of previously passing order-dependent tests failed after applying test prioritisation, 24% after applying test selection and 5% after parallelizing tests.	Order-dependent tests in particular limit the applicability of test acceleration techniques by causing inconsistent outcomes in the test runs they produce.	[91]
🎓 Flaky tests are a threat to the validity of experimental test acceleration algorithms, with researchers taking steps to filter them from their evaluation methodologies.	Researchers should understand that flaky tests are inevitable in real-world test suites and should give forethought to their implications when creating methods that reorder or otherwise modify test suite runs.	[97, 106, 127, 169]
🎓 Flaky tests have been shown to detract from the applicability of mutation testing, fault localisation, automatic test generation, batch testing and automatic program repair .	Approaches for dealing with flaky tests would likely have far-reaching impacts , beyond the obvious benefits of improving test suite reliability.	[108, 114, 138, 139, 157, 167]
🎓 Out of 89 student submissions of a software engineering assignment, 34 contained at least one flaky test.	Flaky tests may hinder software engineering education, which, while representative of industry experience, may place an undue burden upon students.	[140]

2.4.1 Consequences for Testing Reliability

Since the goal of a software test is to provide a signal to the developer indicating the presence of faults, a flaky test that may pass or fail, regardless of program correctness, represents a signal with some amount of noise. In a test suite containing multiple flaky tests, this noise accumulates and may result in an unreliable test suite. Luo et al. [105] explained three deleterious effects that flaky tests can have on the reliability of testing. First, since non-determinism is inherent to test flakiness, attempting to reproduce failures of flaky tests can be more difficult and less valuable. One study empirically evaluated the reproducibility of flaky tests on a large scale [92]. Second, because flaky tests can fail without modifications to the code under test, they can waste developer time via debugging a potentially spurious failure that is unrelated to any recent changes. Multiple studies have examined the manifestation of flaky tests in terms of indicating the presence of non-existent bugs, or conversely, missing real bugs [155, 175]. Lastly, despite flaky tests being an unreliable signal, they may still convey some useful information as to the correctness of the code under test, however, a frequently failing flaky test may eventually be ignored by a developer, thus potentially causing genuine bugs to be missed. To that end, one source found that ignoring flaky test failures may have a negative effect on software stability, measured in the number of crash reports received for the FIREFOX web browser [134].

Reproducibility

Lam et al. [92] posited that, when encountering a failing test during a test suite run, a developer is likely to run that test in isolation in order to reproduce the failure and thus debug the code under test. When trying to reproduce the inconsistent outcome of a flaky test, they demonstrated that this technique may not be effective. They executed the test suites of 26 modules of various open-source Java projects 4,000 times and calculated each test's failure rate — the ratio of failures to total runs. For each test they found to be flaky, meaning a failure rate greater than zero or less than one, they re-executed it 4,000 times again in isolation. They found that, of the 107 tests identified as flaky, 57 gave totally consistent outcomes in isolation. Furthermore, they found that the failure rates of flaky tests appear to differ when executed in isolation. Of the 50 flaky tests that were reproduced in isolation, 19 had lower failure rates and 28 had higher ones. A Wilcoxon signed-rank test demonstrated a statistically significant difference between the two samples, suggesting that the failure rates when executed within the test suite were significantly different to the failure rates when executed in isolation. This result shows that flaky tests appear to behave more consistently when executed in isolation and so this may be a good strategy if trying to reproduce the *failure* of a flaky test, but not its *flakiness*.

Manifestation

Zhang et al. performed an empirical study of test order dependencies [175]. They pointed out that, unlike other sorts of flaky tests, order-dependent tests would only be manifested when the test suite changed by adding, removing or reordering test cases. They described how order-dependent tests can mask bugs, since the order in which the containing test suite is executed might determine whether or not the dependent test is able to expose faults or not. They also explained how test order dependencies might lead to spurious bug reports should the test run order be changed for any reason, potentially manifesting the order-dependent tests and exposing an issue in the test code (e.g., improper setup and teardown) rather than a bug in the code under test, something that a developer may have to allocate time to identify themselves. They went on to search for four phrases related to test order dependency in five software issue tracking systems to understand the manifestations of order-dependent tests in Java projects. In total, they found 96 such tests, 94 of which caused a false alarm, a test failure in absence of a bug, and two caused a missed alarm that was, in other words, a bug in absence of a test failure. The ramifications of a missed alarm are potentially more serious than a false alarm since they may allow a bug to go unnoticed and thus persist after testing.

A later and more general investigation into the prevalence and nature of bugs in test code was performed by Vahabzadeh et al. [155]. Through their manual analysis, they identified 5,556 unique bug reports across projects of the Apache Software Foundation [280], 443 of which they systematically categorised. They categorised these by manifestation, that is, whether the test bug led to a false alarm or to a missed alarm, or a “silent horror” as they termed them, and by their root cause. They found that 97% of test bugs manifested as a false alarm, with test bugs categorised as flaky tests, or due to environmental reasons or mishandling of resources (also types of flaky test according to the definition in Section 2.1), responsible for 53% of these. False alarm test bugs, while potentially less serious than “silent horrors”, can still take a considerable amount of time and effort for developers to debug. Since flaky tests are likely to be the dominant root cause of such bugs this means that they are potentially a significant waste of a developer’s time. Ultimately, the findings of both of these studies suggest that flaky tests are more likely to manifest as a spurious test failure rather than an unnoticed bug, though as the operative word in the latter case suggests, it might simply be that such instances are under reported, because they are, by definition, unnoticed by developers and researchers.

Ignoring Flakiness

Rahman et al. [134] examined the impact of ignoring flaky test failures on the number of crash reports associated with builds of the FIREFOX web browser [221] in both the beta and production release channels. To identify flaky tests, they searched for records of test executions marked with the phrase “RANDOM”, a term commonly used by FIREFOX developers, in the testing logs. In the median case, they found that builds with one or more flaky test failures were associated with a median of 514 and 234 crash reports for the beta and production channels, respectively. Furthermore, a Wilcoxon signed-rank statistical test comparing the numbers of crashes between the two channels showed a statistically significant difference, with Rahman et al. conjecturing that developers were more conservative about releasing production builds with known flaky tests. For those with failing tests in general, these figures were 508 crash reports for the beta channel and 291 for the production channel. In the case of builds with all tests passing, they recorded a median of only two crash reports for each channel. These findings indicate that ignoring test failures, flaky or otherwise, appears to lead to a considerably higher volume of crashes due to missed bugs.

Conclusion for RQ2.1: What are the consequences of flaky tests upon the reliability of testing? One study found that, of 107 flaky tests identified by repeatedly executing their test suites, only 50 could be reproduced as flaky by repeatedly executing them in isolation. These findings suggest that reproducing the flakiness of a flaky test by executing it in isolation may be difficult, though it may still be effective for reproducing the failing case for debugging purposes [92]. In terms of their manifestation, one source identified 94 out of 96 sampled order-dependent tests to have caused a false alarm, meaning that they failed in the absence of a real bug [175]. Another found that 97% of bugs in test code manifested as a false alarm, with flaky tests representing 53% of these [155]. Overall, these results indicate that flaky tests may be a leading cause of test failures that are false alarms. One study found that builds of the FIREFOX web browser with one or more flaky test failures were associated with a median of 234 crash reports in the production channel. This is compared to a median of 291 crash reports associated with builds containing one or more failing tests in general and two reports associated with builds that had all passing tests. These results suggest that ignoring test failures, even if they are flaky, can lead to a higher incidence of crashes [134].

2.4.2 Consequences for the Efficiency of Testing

Projects can benefit from using continuous integration to automate the process of running tests and merging changes [72]. However, since continuous integration may result in a greater volume

of test runs, it can highlight the true extent of flakiness in a test suite [87]. In a continuous integration system, when a developer has been working on their own branch and goes to merge their changes, their code is remotely built and a test suite run is triggered to ensure their changes have not introduced bugs. Should any tests fail, their changes will be rejected. Naturally, given the association between flaky tests and false alarm failures [155, 175], flaky tests can limit the efficiency of the continuous integration process by incorrectly causing builds to fail [33, 86, 88, 89, 110]. In addition, since the majority of software build time is spent on test execution [15], numerous test acceleration approaches have been developed for quickly retrieving the useful information from test runs [78]. Test selection is a technique for speeding up a test run by only executing tests deemed likely to indicate a bug, for example, by only executing those test cases that cover recently modified code [97, 106, 142]. Test prioritisation is used to reorder a test suite such that the test cases likely to reveal faults are executed sooner, achieved by sorting test cases by, for instance, the most modified code elements covered, for [67, 127, 169]. Test parallelisation is an approach for taking advantage of a multi-processor architecture by splitting the execution of a test suite over multiple processors, potentially resulting in significant speedups [15, 22, 66]. Since all of these techniques shrink, reorder or split up a test suite (i.e., change the test execution order), they run the risk of breaking test order dependencies, resulting in inconsistent outcomes for the order-dependent tests that have them [15, 91, 175]. This is problematic since, even though these techniques may reduce testing time, if they cause inconsistent test outcomes then the results of the optimised test run may become unreliable by causing false alarm failures, as previously explained in Section 2.4.1. Studies have shown that flaky tests do impact the results and deployment of these techniques [22, 91, 175], to the extent that researchers have taken steps to filter them as part of their evaluation methodologies [97, 106, 127, 169]. A recent mapping study of test prioritisation in the context of continuous integration found that 13% of sampled publications on the topic identified flaky tests as a problem [131].

Continuous Integration

Lacoste [87], a Canonical developer working on the collaborative development service Launchpad [250], described his experiences of transitioning to a continuous integration system in 2009. His account provided an insight into how the constant execution of tests in a continuous integration system can expose flakiness. He remarked how an intermittent failure caused by a “malfunctioning test”, his description for a flaky test, was a potential reason for a branch being rejected by their old “serial” integration system, thereby causing the team to miss a release deadline. Upon transitioning to a continuous integration system, developers witnessed the magnitude of the flakiness in their test suites, since tests were being executed more often. In particular, he found that the “integration-heavy” tests, that covered multiple processes, were the most problematic ones in the studied test suite.

At a software developed company, called Pivotal Software, a survey deployed by Hilton et al. [71] asked developers to estimate the number of continuous integration builds failing each week due to genuine test failures and those due to flaky test failures. They found no statistically significant difference between the two distributions of estimates after performing a Pearson’s Chi-squared test. This indicated that developers at Pivotal experienced similar numbers of both genuine and flaky failures, something that the manager at Pivotal indicated was a surprising finding.

As part of a study into the cost of regression testing, Labuschagne et al. [86] examined the build history of 61 open-source projects that were implemented in the Java programming language and used the Travis continuous integration service [231]. In each case where a build had previously transitioned through *passed*, *failed* (indicating that the build had failed tests) and back to *passed* again consecutively, they re-executed the build at the failing stage three times. They found that just under 13% of 935 such failing builds in their dataset had failed due to flaky tests.

Durieux et al. [33] performed an empirical study into the prevalence of, and the reasons behind, manually restarted continuous integration builds. Under normal circumstances, a build would be triggered following a code change. If the build fails, due to compilation errors or failed tests, the change is rejected. If a developer decides to manually trigger a build, without making any changes,

it is referred to as a *restarted build*. They analysed the logs from over 75 million builds on the Travis system. They found that 47% of previously failing builds that were restarted subsequently passed, indicating that they were affected by some non-deterministic behaviour, since no code change was involved. They went on to identify inconsistently failing tests to be the main reason for developers restarting builds. Furthermore, they found that projects with restarted builds experienced a slow down of their pull request merging process of 11 times compared to projects without restarted builds. This result suggests that flaky tests have the potential to hinder the continuous integration process by spuriously failing builds and requiring manual developer intervention.

Additionally, Memon et al. [110] performed a study into reducing the cost of continuous testing at Google. They described flaky tests as a reality of practical testing in large organisations, citing them as a constraint to the deployment of the results of their work. Specifically, they identified approximately 0.8% of their total dataset of over five million “test targets”, a term used by Google to describe a buildable and executable code unit labelled as a test, as flaky. Of the 115,160 test targets that had historically passed at least once and failed at least once, flaky tests constituted 41%.

Microsoft’s distributed build system, CLOUDBUILD, was the object of analysis in a study by Lam et al. [88]. By examining the logs from repeated test executions, these authors identified 2,864 unique flaky tests across five projects. Overall, of the 3,871 individual builds that they sampled, 26% presented flaky test failures. Furthermore, data from FLAKES, the flaky test management system integrated into CLOUDBUILD, was used in a later study [89]. Over a 30-day period, the authors found that FLAKES identified a total of 19,793 flaky test failures across six subject projects, representing just 0.02% of the sampled test executions. However, had the FLAKES tool not identified such cases, they reported that flaky test failures would have been responsible for a total of 1,693 failed builds, which would have represented 5.7% of all the failed builds sampled. This finding suggests that, even when flaky tests are not particularly prevalent, the percentage of builds that may be affected by flaky test failures can be relatively significant.

Test Acceleration

As part of an empirical study of test order dependence, Zhang et al. [175] assessed how many test cases gave inconsistent outcomes when applying five different test prioritisation schemes across the test suites of four open-source Java projects. Their results showed that, for one subject in particular with 18 previously identified order-dependent tests, up to 16 gave a different outcome compared to a non-prioritised test suite run. This finding indicated that the soundness of test prioritisation was, at least for this project, significantly impacted by order-dependent tests — even though test prioritisation methods are not supposed to affect the outcomes of tests at all.

A later evaluation into the impact of test parallelisation on test suites for Java programs was conducted by Candido et al. [22]. They measured the speedup of the test suite run time and the percentage of previously passing tests which now failed, indicating flakiness, under five parallelisation strategies across 15 subjects. Their results showed that flaky tests were manifested within four projects under every strategy and that 11 projects had totally consistent outcomes for at least one. Their results also indicated that the finer-grained techniques that involved parallelisation at the test method level, as opposed to just the test class level, manifested the most flaky tests. These techniques were also the most effective, indicating that the speedup achieved and the reliability of the test outcomes could be a trade-off for developers to consider when order-dependent tests are present in the test suite.

Lam et al. [91] evaluated the consequences of order-dependent tests upon the soundness of regression test prioritisation, selection and parallelisation. They evaluated a variety of algorithms upon 11 modules of open-source Java projects with both developer-written and automatically generated test suites with the order-dependent tests already identified in previous work [90]. After applying test prioritisation, they found a total of 23% of order-dependent tests in their subjects’ developer-written test suites failed and 54% of such tests in their automatically generated test suites failed, when they were previously passing. For test selection, this was 24% and 4% for developer and automatic test suites respectively and, for parallelisation, 5% and 36%, respectively.

Several studies have revealed that researchers are aware of the negative impacts that flaky tests can have on attempts to improve testing efficiency, as they often take steps to remove them in their evaluation methodologies. Leong et al. [97] compared the performance of several test selection algorithms using one month's worth of data from Google's Test Automation Platform. They demonstrated that, for one particular algorithm that prioritised tests based on how frequently they had historically transitioned between passing and failing, its performance was better when evaluated without flaky tests. Specifically, when evaluated with a dataset containing flaky tests, for up to 3.4% of historical commits, the algorithm did not select one or more tests whose outcome had changed, compared to up to 1.4% when flaky tests were filtered out. Since the goal of test selection is to only execute informative tests, by missing more tests whose outcome had changed (suggesting that a particular commit may have introduced/fixed a bug), the algorithm performed worse at its intended purpose. They used this as justification for filtering flaky tests in their wider evaluation methodology. Similarly, Peng et al. [127] found that, when comparing test prioritisation approaches based on information retrieval techniques, they mostly appeared to perform better when evaluated with a dataset filtered of flaky tests.

When evaluating a data-driven test selection technique deployed at Facebook, Machalica et al. [106] remarked how flaky tests represented a significant obstacle to the accuracy of their approach. Their test selection technique used a classification algorithm, that was trained with previous examples of failed tests and their respective code changes, to only select tests for execution when it predicted that they might fail given a new code change. Finding that the set of flaky tests in their dataset was four times larger than the set of deterministically failing tests, they explained that if they did not filter flaky tests from their training data then they would run the risk of training their classifier to capture tests that failed flakily rather than those that failed deterministically due to a fault introduced by a code change. Yu et al. [169] faced similar problems with their machine-learning-driven approach for test case prioritisation in the context of user interface testing. They identified flaky tests as a threat to the internal validity of their research, admitting that they did not filter flaky tests from their dataset and thus could have limited the accuracy of their trained model. As future work, these authors planned to explore ways to tackle test flakiness.

Conclusion for RQ2.2: What are the consequences of flaky tests upon the efficiency of testing? One study found that 13% of failing builds across 61 projects using the Travis system, which had previously passed and then immediately went on to pass again, were caused by flaky tests [86]. Similarly, of a sample of over 75 million builds, 47% were manually restarted builds that previously failed and subsequently passed, indicating that they were affected by non-deterministic behaviour such as flaky tests [33]. At Google, one study found that 41% of "test targets" that had previously passed and failed at least once were flaky [110]. At Microsoft, another study identified flaky test failures within 26% of all the builds they sampled [88]. Another Microsoft study demonstrated that the 0.02% of flaky test failures from over 80 million test executions, over a 30-day period, could have been responsible for what would have been 5.7% of all the failed builds in that period, had they not been identified [89]. These results indicate that flaky tests can limit the efficiency of continuous integration by spuriously failing builds, thereby requiring manual intervention from developers. Evaluating several test parallelisation strategies across 15 projects, one study found that order-dependent flaky tests were manifested within 11 of the 15 under at least one strategy [22]. Across a range of developer-written test suites with previously identified test order dependencies [90], another source found that 23% of order-dependent tests failed, when previously passing, after applying test prioritisation, 24% after applying test selection and 5% after parallelizing tests [91]. These findings suggest that order-dependent tests in particular are an obstacle to the applicability of test acceleration techniques. Flaky tests may also hinder the formulation and evaluation of new test acceleration methods. One study found that when evaluating a transition-based test selection algorithm, it failed to perform correctly in 3.4% of cases when flaky tests were present in the evaluation dataset, compared to 1.4% once they were removed [97]. Other sources have described how

flaky tests were a threat to the validity of their techniques, with most opting to filter them as a part of their methodologies [106, 127, 169].

2.4.3 Other Consequences

Beyond the general consequences for testing already examined in this section, various sources have demonstrated that flaky tests can pose a considerable problem to a diverse range of specific testing activities. The effectiveness of techniques such as mutation testing [76, 139], fault localisation [157], automatic test generation [125, 138], batch testing [114], and automatic program repair [108, 167] have all been shown to be impacted by the presence of test flakiness. Furthermore, in the domain of software engineering teaching, studies have revealed that flaky tests can detract from the educational value of various teaching activities and possibly become a burden for students [132, 140, 146].

Mutation Testing

Mutation testing is the practice of assessing a test suite’s bug finding capability by generating many versions of a software under test with small syntactical perturbations, to resemble bugs, known as *mutants* [75]. A test is said to *kill* a mutant if it fails when executed upon it, thus witnessing the artificial bug. If a test covers mutated code but does not kill it then the mutant is said to have *survived*. At the end of a test suite run, the percentage of killed mutants is calculated and is called a *mutation score*, commonly used as a measurement of test suite quality. Demonstrating the extent to which flakiness can impact mutation testing, Shi et al. [139] conducted an experiment with 30 Java projects. In the context of this study, the flakiness they were specifically referring to relates to non-determinism in the coverage of tests rather than necessarily the test outcome. As an initial motivating study, they examined the number of statements that were inconsistently covered across their subjects when repeatedly executing their test suites. Overall, they found that the coverage of 22% of statements was non-deterministic. They used a mutation testing framework, called PIT [262], to generate mutants for these same subjects. The framework first runs the test suite to analyse each test’s coverage so that it generates mutants that the tests have a chance of killing. Given the level of flakiness in the coverage of these tests, however, the authors explained that when they are executed again to measure mutant killing, they may not even cover their mutants, such that their killed status is unknown. Investigating this phenomenon’s extent by applying PIT to their subject set, they found that over 5% of mutants ended up with an unknown status due to inconsistent coverage, leading to uncertainty in the mutation score. Assuming that all unknown mutants were killed, the overall mutation score would have been 82% and — further assuming that they all survived — this score would have been 78%, showing that non-deterministic coverage can limit the reliability of this test effectiveness metric.

Fault Localisation

Vancsics et al. [157] studied the influence of flaky tests on automated fault localisation techniques. Fault localisation uses the outcomes and coverage of tests to identify the location of faulty program elements. This is motivated by the reasoning that a group of program statements, for example, that is disproportionately covered by failing tests is likely to contain a bug. Fault localisation techniques associate program elements with *suspiciousness rankings*, calculated in various ways [166], which capture the probability that they contain a fault. As the subjects of their investigation, they took multiple versions of a single project from the *Defects4J* dataset, each version containing a reported bug, and evaluated the effectiveness of three popular fault localisation techniques in locating each respective bug. They executed the test suites of each buggy version 100 times, collecting test outcomes and method-level coverage data. To simulate the presence of flaky tests, they repeatedly invoked each studied technique to compute suspiciousness rankings for each method, increasingly adding noise to the test outcomes. Specifically, they artificially changed test outcomes from pass to fail, or vice versa, at random with an increasing probability, up until the point where each

test appeared to pass or fail with a 50/50 ratio. This required them to modify the suspiciousness ranking formulae of each technique to consider a test outcome as a probability of passing or failing rather than a binary value, since typically fault localisation does not require multiple test suite runs and would thus only have one recorded outcome for each test. To measure the impact of the artificial flaky tests, they measured the extent to which the suspiciousness rankings changed with increasing flakiness. Their results indicated that, in general, each technique was affected by increasing flakiness. In other words, the magnitude of the difference in the suspiciousness rankings of each technique for many of the subject bugs was positively correlated with the magnitude of the artificial flakiness.

Automatic Test Generation

Shamshiri et al. [138] studied the effectiveness of automatically generated test suites for the testing of Java programs. They applied three unit test generation tools, RANDOOP [268], EVOSUITE [219], and AGITARONE [199], to a dataset of over 300 faults across five open-source projects, assessing how many bugs the automatically generated unit tests could detect. Through this process they also identified the number of flaky tests that were generated by each of the three tools. Of the tests generated by RANDOOP, which uses feedback-directed random test generation, an average of 21% exhibited non-determinism in their outcomes. The EVOSUITE tool, which leverages a genetic algorithm, produced flaky tests at an average rate of 3%. Only 1% of the tests generated by the commercial, proprietary tool called AGITARONE were found to be flaky. These findings demonstrate that automated tools, as well as developers, are also capable of producing flaky tests, threatening the overall reliability of automatically generated test suites.

A later study by Paydar et al. [125] examined the prevalence of flaky tests within the regression test suites generated specifically by RANDOOP. They explained how regression test suites are useful in as far as they capture the behaviour of a program at a given point in time and are used to identify if a recent code change has any unintended effects. If a regression test suite contains flaky tests then it does not accurately reflect the behaviour of the program and is thus less informative for developers. These authors took between 11 and 20 versions of five open-source Java projects and used RANDOOP to generate regression test suites, which were the main objects of analysis. Overall, they found that 5% of their automatically generated test classes were flaky, and on average, 54% of the test cases within each of these were flaky, further demonstrating that automatically generated tests can contribute to the flakiness of a test suite.

Batch Testing

Najafi et al. [114] investigated the impact of flakiness on batch testing, a technique for improving efficiency within a continuous integration pipeline. Instead of testing each new commit individually, batch testing groups commits together to reduce test execution costs. Should the batch pass then each commit can proceed to the next stage of the pipeline. Should a batch fail, it has to be repeatedly bisected to identify the commit(s) that caused the failure, effectively performing a binary search for the culprit. They explained that flaky tests are a threat to the applicability of batch testing, since a spurious flaky failure may lead to unnecessary bisections. To mitigate this, smaller batch sizes can be selected since the flaky failure will affect fewer commits, though this limits the potential efficiency gains from applying batch testing in the first place. An evaluation upon three unnamed projects at Ericsson demonstrated the following negative correlation: the more flakiness within a test suite, the smaller the most cost-effective batch size.

Automatic Program Repair

Test suite-based automatic program repair is a family of techniques for automatically generating patches for bugs [95]. The test suites of the subject programs are used to assess the correctness of the patch, such that if all the tests pass then the patch is deemed suitable. Like any technique based on the outcomes of tests, automatic program repair is sensitive to flaky tests since they introduce

unreliability into this assessment. Specifically, a flaky test failure could lead to a correct patch being spuriously rejected [108]. Ye et al. [167] presented an empirical evaluation of automated patch assessment, analysing 638 automatically generated patches. Specifically, they employed a technique known as *Random testing based on Ground Truth* (RGT) to determine if a generated patch was correct or *overfitting*. A patch is considered overfitting if it passes the developer-written tests it was generated from, yet is generally a poor solution to the bug in question and may fail on tests held out from the patch generation process. To that end, the RGT technique uses an automatic test generation technique such as Randoop [125] or EvoSuite [40, 44] to assess the generated patches. If a patch fails an automatically generated test, it is labelled as overfitting. In this case, the test is also applied to a developer-written, *ground truth* patch to determine the behavioural difference with the overfitting patch. Given the potential for automatic test generation techniques to produce flaky tests [125, 138], the RGT technique includes a preprocessing stage where it repeatedly executes each generated test on the ground truth patch. Given that the ground truth patch is considered to be correct, any test failure at this stage is considered to be flaky and the test is discarded. Having discarded 2.2% of tests generated by EvoSuite and 2.4% generated by Randoop for this reason, Ye et al. concluded that flaky test detection is an important consideration for researchers in automatic program repair. The authors remarked how this was a threat to the internal validity of their study, explaining how the flaky tests they discarded may still have exposed behavioural differences between generated and ground truth patches. As such, this could have led to them underestimating the effectiveness of RGT in the context of automated program repair.

Teaching and Education

Using a contrived electronic health record system developed and tested by students of a software engineering course as an object of study, Presler-Marshall et al. [132] analysed the effect of various factors on the flakiness of user interface tests within web apps. As part of their motivation, they explained that, while it may well reflect real industry experiences, ambiguous feedback from flaky tests can introduce undue stress to students.

Shi et al. [140] investigated the impact of assumptions about non-deterministic specifications upon the incidence of flaky tests in both open-source Java projects and student assignments. They described how the student assignments in software engineering courses are typically graded with the assistance of automated tests. If it turns out that these tests are flaky, the authors explained, then incorrect solutions may have passing tests and, visa versa, correct solutions may have failing tests and thus may result in unfair grades being awarded. They went on to evaluate the incidence of flaky tests in student submissions of a software engineering assignment, where students were asked to implement a program and its test suite, and found 110 flaky tests across 89 submissions, with 34 containing at least one flaky test.

A study by Stahlbauer et al. [146] introduced a formal testing framework for the educational programming environment called Scratch [273]. The authors instantiated this framework with the WHISKER tool, providing automatic and property-based testing for Scratch programs. Since Scratch is an educational platform, they reasoned that it would add educational value by assisting learners to identify functionality issues in their programs. Upon evaluation of their tool, they found that just over 4% of the combinations of tests and projects to exhibit flakiness if SCRATCH's underlying random number generator was not seeded. This result suggests that flaky tests have the potential to confound their framework and thus potentially detract from its intended educational value.

<p>Conclusion for RQ2.3: What are the other consequences of flakiness? Tests with flaky coverage have been shown to be deleterious to the effectiveness of mutation testing, with one experiment finding that 5% of mutants ended up with an unknown killed status, resulting in uncertainty around the final mutation score [139]. One source has demonstrated that fault</p>

localisation may be sensitive to the inconsistent outcomes of flaky tests, finding the applicability of three popular techniques to be negatively correlated with the magnitude of simulated flakiness [157]. Automatic test generation was shown to have the potential to produce flaky tests, in particular the Java tool RANDOOP, where one study found 21% of the overall tests it produced across five subjects to be flaky [138]. The potential efficiency benefits of the batch testing technique was also shown to be negatively affected by test flakiness, as evaluated with three proprietary subjects [114]. As another technique dependent on the outcomes of tests, one study evidenced the negative impacts of flaky tests on test suite based automatic program repair, specifically the automatic assessment of the generated patches [167]. Studies have also found that flaky tests can be identified within students’ submissions of software engineering assignments, potentially impacting grading [140]. Their authors’ suggested that they can become an obstacle to learning [146] and also place undue stress on students [132].

2.5 Detection

Armed with knowledge of their underlying causes, as covered in Section 2.3, and motivated by their negative impacts, as described in Section 2.4, I now consult sources presenting insights and strategies for detecting flaky tests. The automatic identification of flakiness in a test suite has the potential to be useful for developers by not only highlighting test cases that may need to be rewritten, but also by enabling the use of various mitigation strategies (see Section 2.6.1). As in Section 2.3, I separately examine techniques targeting order-dependent flaky tests, which given their unique costs, have been afforded special attention in the literature. I answer **RQ3** by following the same pattern as the previous sections, posing and answering two sub-research questions, with Table 2.8 providing a summary.

RQ3.1: What techniques and insights are available for detecting flaky tests? This captures the findings and approaches regarding the detection of flaky tests. My answer provides a tour of the relevant literature and demonstrates the diversity of techniques available, ranging from those that require executions of a test suite to those that are entirely static in their analyses. Beyond presenting specific automatic techniques, I also examine general approaches for flaky test detection, as well as offering advice for how best to apply automated tools. (See Section 2.5.1.)

RQ3.2: What techniques and insights are available for detecting order-dependent tests? This demonstrates attempts to identify test order dependencies specifically, motivated by the specific impacts and causes of this category of flaky tests. I provide a comparative review of such techniques and demonstrate an emerging line of work. (See Section 2.5.2.)

Table 2.8: Summary of the findings and implications answering **RQ3**: What insights and techniques can be applied to detect flaky tests? Relevant to researchers (☞) and developers (</>).

Finding	Implications	Source
☞ Almost all flaky tests were independent of the execution platform , meaning, for example, that they could be flaky under multiple operating systems — even if they were dependent on the execution environment.	Techniques for detecting flaky tests ought to consider environmental dependence with a higher priority than platform dependence.	[105]
☞ Of the <i>asynchronous wait</i> flaky tests, 34% used a simple time delay to enforce a particular execution order .	A considerable portion of asynchronous wait flaky tests may be manifested by changing — in particular by decreasing — a simple time delay .	[105]
☞ The vast majority of flaky tests of the <i>concurrency</i> category involved, or could be reduced to, only two interacting threads , and 97% pertained to concurrent access to in-memory resources only.	Previous approaches to increasing context probability [38] may be applicable to detecting concurrency related flaky tests.	[105]

<p>🎓 Dependency on external resources was blamed for 47% of flaky tests of the <i>test order dependency</i> category.</p>	<p>Not all order-dependent tests can be detected by considering the internal state of in-memory objects; modelling of the external environment may be required. [105]</p>
<p>🎓 Repeating tests as a method of identifying flakiness is a common practice. Between 15% and 18% of test methods were found to be flaky in open-source Java projects using this approach.</p>	<p>Repeating tests can be considered a reliable baseline for detecting flaky tests since it directly witnesses inconsistent outcomes. It can also become very costly in terms of execution time when performing many repeats. [16, 89, 92]</p>
<p></> One study found that 88% of flaky tests were found to consecutively fail up to a maximum of five times before passing, though another reported finding new flaky tests even after 10,000 test suite runs.</p>	<p>There appears to be no clear, optimum number of reruns for identifying flaky tests. [8, 92]</p>
<p>🎓 By identifying tests that cover unchanged code but whose outcome changes anyway, DEFLAKER was able to detect 96% of flaky tests previously identified by repeatedly executing them up to 15 times.</p>	<p>By taking differential coverage into account, an automated tool can identify the vast majority of flaky tests with only a single test suite run. [16]</p>
<p></> By randomizing the implementations of non-deterministic specifications, NONDEX found 60 flaky tests across 21 open-source Java projects.</p>	<p>Developers should take care when dealing with unspecified behaviour, such as the iteration order of unordered collections, so as not to introduce flaky tests. [61, 140]</p>
<p></> The addition of CPU and memory stress during test suite reruns was shown to increase the rate at which flaky tests were detected.</p>	<p>Since stress-loading tools are readily available, this technique is easily accessible to developers and could reduce the runtime cost of detecting flaky tests. [144]</p>
<p>🎓 By fitting a probability distribution over the inputs evaluated in assertion statements and estimating the probability that they would fail, FLASH identified 11 new flaky tests in machine learning projects and verified a further 11 previously fixed flaky tests.</p>	<p>Despite the differences in the common causes of flaky tests in machine learning projects (see Section 2.3.3), specific approaches for detecting them have demonstrated some success in this domain. [35]</p>
<p>🎓 Through static pattern matching within test code, 807 instances of timing dependencies potentially indicative of flaky tests of the asynchronous wait category were identified across 12 projects. A sample of 31 of these were all identified as true positives via manual analysis.</p>	<p>This technique could prove useful as a first indicator of potential flakiness, given that it requires no test executions, but is naturally limited by the fact that it cannot verify that the tests it identifies are genuinely flaky. [161]</p>
<p>🎓 The application of machine learning techniques to the detection of flaky tests has shown promising results, with one study reporting an overall F1 score of 0.86 across 24 open-source Java projects using a model based on features regarding particular identifiers in test code, the presence of test smells and other test characteristics.</p>	<p>Since these techniques do not require repeated test suite runs and are not tied to a specific programming language, they are potentially very useful to developers, who may welcome a more general-purpose technique and may not have the time or resources to repeatedly rerun their test suites. [8, 65, 127, 159]</p>
<p>🎓 Of 245 flaky tests, 75% were flaky from the commit that introduced them.</p>	<p>Running automatic flaky tests detection tools only after introducing tests may identify the majority of cases, but will exclude a significant portion. [93]</p>

<p>Of 61 flaky tests that were not flaky from their inception, the median number of commits between the commit that introduced them into the test suite and the commit that introduced their flakiness was 144.</p>	<p>Periodically running flaky tests detection tools after larger numbers of commits, i.e., 150 or so, is likely to achieve a good cost-to-detection ratio, as opposed to running them after every commit, which may be prohibitively expensive. [93]</p>
<p>Executing a test suite in reverse was able to manifest 72% of all the developer-written and automatically generated order-dependent tests that were identified as part of a larger evaluation of several strategies within DTDETECTOR. This approach was also the fastest by at least one order of magnitude.</p>	<p>Executing tests in their reverse order and identifying order-dependent tests via inconsistent outcomes compared to their original order is a fast and reasonably effective baseline approach. [175]</p>
<p>By monitoring reads and writes to Java objects between test runs, ELECTRICTEST detected all the same order-dependent tests in a single instrumented test suite run as DTDETECTOR would when it repeatedly executes the tests. It also detected hundreds more that were not verified.</p>	<p>Techniques of instrumenting objects to identify potential test order dependencies can be very efficient since they require only a single test suite run. Yet, they cannot verify that a given order-dependent test is manifest and so could be considered prone to false positives and a low precision, even if they have a high recall. [15]</p>
<p>By executing minimal test schedules to verify possible order-dependent tests, PRADET filters false positives detected using an object instrumentation technique similar to ELECTRICTEST. However, it detected fewer order-dependent tests overall than simple techniques based upon reversing or shuffling the test run order.</p>	<p>Despite their simplicity, techniques for detecting order-dependent tests based upon re-executing whole test suites in different orders may be more efficient and effective than more sophisticated approaches. [47]</p>
<p>Filtering possible order dependencies that were unlikely to be manifest, using natural language processing techniques upon the leading verb and nouns of test case names, decreased the run time of TEDD by between 28% and 70% when detecting order-dependent tests.</p>	<p>There appears to be valuable information regarding the existence of test order dependencies present in the names of test cases, which may be extracted using natural language processing techniques. [17]</p>
<p>An evaluation of IDFLAKIES found that, of 422 identified flaky tests, 50% were order-dependent.</p>	<p>The prevalence of order-dependent tests as reported by automatic tools appears to be higher than as reported by developers (see Table 2.5), suggesting that developers may be unaware of many order-dependent tests or possibly do not consider them a priority for repair. [90]</p>
<p>A chi-squared statistical test of independence revealed that 70 out of 96 flaky tests had different failure rates when executed in different test class orders, to a statistically significant degree.</p>	<p>The binary distinction of flaky tests into order-dependent or not may be overly coarse, since the probability that a flaky test fails may be dependent on the test run order, yet still be neither exactly zero or one. [92]</p>

2.5.1 Detecting Flaky Tests

Studies have explored the many different ways of identifying flaky tests within test suites. One paper offered insights on how best to manifest flaky tests derived from inspecting previous repairs of flaky tests within the commit histories of a multi-language sample of projects [105]. With regards to automatic techniques, a common baseline approach is to repeatedly execute the tests in

an attempt to identify inconsistencies in their outcomes [16, 89, 92, 187]. More targeted approaches have considered test coverage [16] or the characteristics of specific categories of flakiness [35, 140, 144]. Several references have presented approaches that do not explicitly require test reruns, such as those based on pattern matching [161] or machine learning [8, 82, 128, 159]. Given the potentially prohibitive costs of repeatedly applying automatic flaky test detection tools, one reference experimentally examined when it was best to use them in order to achieve a good ratio of detection to run time cost [93].

Insights from Previous Repairs

By studying historical commits that repaired flaky tests in open-source projects of the Apache Software Foundation, Luo et al. [105] offered several insights into how best to manifest test flakiness. They found that 96% of the flaky tests they examined were independent of the execution platform, meaning that they could be flaky on different operating systems or hardware, even if they were dependent on a component in the execution environment such as the file system. From this finding, they suggested that techniques for detecting flaky tests ought to consider environmental dependence with a higher priority than platform dependence. Of the flaky tests they categorised as being of the *asynchronous wait* category, they found that 34% used a simple time delay or a `sleep` or `waitFor` method [263, 277], to enforce a certain ordering in the test execution. They explained that these particular cases of flakiness may be manifested by changing, specifically decreasing, this time delay. They suggested a second manifestation approach for tests in this category: adding an additional time delay somewhere in the code. This technique, they explained, could be applicable to the 85% of sampled flaky tests in this category that neither depended upon external resources (since they are harder to control) and involved only a single ordering (one thread or process waiting on one other thread or process). On this theme, Endo et al. [39] proposed a technique for manifesting race conditions in JavaScript applications, which are typically highly asynchronous, by selectively introducing delays between events. Following an empirical study, they found their technique was also capable of manifesting flaky tests of the *asynchronous wait* category, identifying two such instances among 159 test cases. Of the flaky tests Luo et al. categorised as being of the *concurrency* category, they found that almost all involved, or could be simplified to, two interacting threads and that 97% were related to concurrent access to in-memory objects, as opposed to being related to external resources such as the file system. This, the authors posited, implies that existing techniques [38] for increasing context switch probability could manifest this kind of test flakiness. With regards to test order dependencies, the authors found that 47% of the flaky tests that they categorised under this cause were facilitated by external resources and thus recording and comparing internal object states may be insufficient to detect these instances, instead requiring modelling of the external environment or reruns with different test run orders.

Repeating Tests

The most straightforward approach to detecting flaky tests is to repeatedly execute the tests of a test suite. This is done with the rationale that, if re-executed enough times, the inconsistent outcomes of any flaky tests will be manifested. This approach has been universally adopted, becoming a part of the testing infrastructure within large software companies such as Google [187] and Microsoft [89]. Furthermore, plugins and extensions are available for popular testing frameworks, such as SUREFIRE [254] for MAVEN [253] and FLAKY [206] for PYTEST [266], for repeatedly executing failing tests to identify flakiness. Due to its ubiquity and simplicity, the repeated execution of tests may be considered a baseline approach from which to evaluate more sophisticated methods for detecting flakiness [16].

Bell et al. [16] assessed how many flaky tests could be identified in open-source Java projects by repeating the failing tests until they either passed, thus indicating a flaky test via an inconsistent outcome, or until a limit was reached. Reasoning that repeatedly executing a test case using the same strategy may result in it failing for the same reason each time and thus not exposing a flaky outcome, they used an incremental approach combining three rerunning strategies. They

first repeated a failing test up to five times within the same Java Virtual Machine (JVM) process, followed by up to five times within separate processes and then again up to five times, cleaning the execution environment and rebooting the machine between repeats. The authors applied this approach to 28,068 test methods across 26 projects and found 18% of the test cases in their dataset to be flaky. Of these flaky tests, 23% were manifested by re-execution within the same process, a further 60% required repeats within their own JVM instances and the remainder could only be identified by removing generated files and directories (by using Maven’s `clean` command [253]) and then rebooting the machine.

Deciding how many times to repeat a test is a difficult one, since a low number may risk missing the more elusive flaky tests and a high number can impose significant runtime costs. For example, at Google, failing tests may be repeated up to ten times to identify if they are flaky [187]. Whereas Microsoft’s FLAKES system repeats failing tests only once by default [89]. Lam et al. [92] set out to identify what would be a good number of repeats by examining the lengths of sequences of consecutive failures when repeatedly executing flaky tests, which they referred to as *burst lengths*. Initially, they re-executed 4,000 times, in various orders, the test suites of 26 modules of open-source Java projects, consisting of 7,129 test methods, to identify the flaky tests. They identified 107 flaky tests this way and went on to examine the cumulative distribution function of their *maximal* failure “burst lengths”. The authors found that around 88% of the flaky tests in their study would fail up to five times consecutively before passing and revealing flakiness. This led them to suggest that no more than five repeats should be necessary to manifest the vast majority of flaky tests.

Kowalczyk et al. [84] presented a mathematical model for ranking flaky tests by their severity, based on their outcomes following repeated executions. Their model includes the concept of a *version*, a series of test runs where the source code, program data, configuration, and any other artifact that may be expected to affect test outcomes, remains unchanged. Within a single version, the authors described two flakiness measures. The first is the *entropy* of the test, a value between 0 and 1 inclusive where 0 indicates a test that always passes or always fails and 1 indicates a test that passes half the time and fails the other half. For a test with a probability of passing in a given version p , this is calculated as $-p \log_2 p - (1 - p) \log_2 (1 - p)$. Their second measure, the *flip rate*, captures the rate at which the test transitions between passing and failing, or vice versa, and is calculated simply as the ratio of the number of such transitions over the maximum number possible, which would simply be one less than the number of repeats. To aggregate these scores for a test across versions, the authors propose an *unweighted model* — simply the arithmetic mean of the score across some number of previous versions — and a *weighted model* — an exponentially weighted, moving average that gives more weight to more recent versions. Leveraging data from two services at Apple, the authors also explain how to use this generalisable model for test flakiness to both identify flaky tests and rank them according to their severity.

Differential Coverage

Bell et al. [16] presented an approach for detecting flaky tests in Java projects without having to repeatedly execute them. Their technique is based on the notion that if a test previously passed and now fails, and does not cover any code that was recently changed, then the failure must be spurious and thus flaky. To that end, they developed a hybrid statement and class-level instrumentation technique that considers differential coverage only, meaning the statements that have recently changed according to a version control system, to identify if a given test method does indeed cover changed code or not. They implemented their technique as a tool named DEFLAKER and evaluated it using 5,966 commits across 26 open-source Java projects. Initially, they compared their approach to a three-tiered strategy of rerunning failing tests until they passed, thus identifying flakiness, or until an upper limit was reached. They found that DEFLAKER was able to identify 96% of the flaky tests identified by rerunning tests in this way, demonstrating its comparable effectiveness with this baseline approach. A further experiment demonstrated that, out of 96 flaky tests confirmed as genuine based upon historical fixes from version control data, DEFLAKER was able to verify up to 88%. Since DEFLAKER is a coverage-based tool with no

requirement to repeatedly execute tests, it has the advantage of being significantly more efficient. Examining the cost of DEFLAKER’s instrumentation, the authors measured the time overhead compared to several popular coverage measurement tools. They found that DEFLAKER incurred a mean increase of approximately 5% the un-instrumented run time of the test suite, compared to a range of 33% to 79% for the other tools, concluding that DEFLAKER was lightweight enough for continuous use during testing.

Non-deterministic Specifications

Shi et al. [140] presented a technique for identifying test flakiness stemming from the assumption of a deterministic implementation of a non-deterministic specification, or *ADINSs* as they abbreviated them, within Java projects. Such specifications are “underdetermined” and leave certain aspects of behaviour undefined and thus up to the implementation, potentially allowing for multiple correct outputs for a single input [60]. They went on to describe three categories of ADINSs. The first was *random* and relates to the case where code assumes that the implementation of a method deterministically returns a particular scalar value when it is not specified to do so. They give the `hashCode` method of the abstract `Object` class [258] as an example of a potential subject of such an ADINS, since it is not specified to return any particular integer value and thus may be highly implementation dependent. The second category was *permute* and describes ADINSs regarding a particular iteration order of a collection type object when it is left unspecified. For example, a test that assumes a `HashMap` [229] will iterate over its elements in a particular order would be guilty of this type of ADINS and could thus be flaky across platforms, where the implementation of `HashMap` may vary. The final category was *extend* and describes the situation where a specification gives the length of a collection object returned by some method as a lower bound and the assumption is made by the developer that the object will *always* be of that length. They gave the `getZoneStrings` of the `DateFormatSymbols` class [213] as an example, since it is specified to return an array of length at least five, which it does in Java 7, but returns one of length seven in Java 8. If a developer writes tests that expect this method to return an array of length exactly five then they may fail when transitioning from Java 7 to Java 8, becoming a flaky test of the *platform dependency* category in Table 2.4.

In order to manifest instances of ADINSs, they developed a tool named NONDEX (also the subject of its own paper [61]) that consisted of a re-implementation of a range of methods and classes in the Java standard library which randomised the non-deterministic elements of their respective specifications. For example, the implementation of `HashMap` was changed such that the iteration order could be randomised each time to deliberately violate and manifest any permute ADINSs. Applying their tool, they were able to identify up to 60 flaky tests across 21 open-source Java projects. Furthermore, they executed their approach upon student submissions of an assignment for a software engineering course that required students to both implement and write tests for a library management application. They identified up to 110 tests with ADINSs that were therefore flaky under their randomised implementations.

Developing the same theme, Mudduluru et al. [112] devised and implemented a type system for verifying determinism in sequential programs. Their Java-targeting implementation consisted of a type checker and several type annotations including `@NonDet`, `@OrderNonDet` and `@Det`. Respectively, these indicate a non-deterministic expression, an expression evaluating to a collection type object with a non-deterministic order (e.g. an instance of `HashMap`), and an entirely deterministic expression. Their type checker verifies these manually-specified determinism constraints and was demonstrated to expose 86 determinism bugs across 13 open-source Java projects. While not specifically targeting flaky tests, the authors demonstrated that their type checker was able to identify determinism bugs that NONDEX was not, thus suggesting that it may be useful for identifying flaky tests, albeit indirectly.

Noisy Execution Environment

Silva et al. [144] presented a technique, called SHAKER, for automatically detecting flaky tests, particularly of the *asynchronous wait* and *concurrency* categories. Their technique uses a stress-loading tool to introduce CPU and memory stress while executing a test suite, with the rationale being that this stress will impact thread timings and interleaving, potentially manifesting more flaky tests than if the test suites were just repeatedly executed as normal. The authors took 11 Android projects and repeatedly executed their test suites 50 times to arrive at an initial dataset of known flaky tests. Finding 75 flaky tests with this method, they split these into a “training” set, containing 35, and a “testing” set, containing 40. They used the training set to search for the set of parameters to the stress-loading tool that manifested the most flaky tests. After identifying the best parameter set, the authors applied SHAKER to the remaining 40 flaky tests to identify how many it could identify. They compared this with rerunning the respective test suites without stress, as they did to find the initial set of previously known flaky tests. Their results indicated that SHAKER was able to detect flaky tests at a faster rate than the standard rerunning approach. In particular, 26 of the 40 flaky tests failed after just a single run with the SHAKER tool.

Machine Learning Applications

Dutta et al. [35] presented their FLASH technique for identifying flaky tests specific to machine learning and probabilistic projects. The reasoning underpinning their approach is that machine learning algorithms are inherently non-deterministic and operate probabilistically, hence their outputs ought to be thought of as probability distributions as opposed to deterministic values. Their technique consists of mining a test suite for approximate assertions, such as those that check that the output is within a certain range or approximately equal to some expected value with some specified tolerance. It then instruments the associated test cases and repeatedly executes them to arrive at a sample of actual values evaluated within each mined approximate assertion. To determine if the sample of values is large enough, it uses the *Geweke diagnostic*, which is satisfied once the mean of the first 10% of samples is not significantly different from the final 50% within some specified threshold. Once FLASH has collected enough samples for each assertion, it fits an *empirical distribution* over each of them, used to calculate the probability of the assertion failing. Under their technique, a test is considered flaky if it has an inconsistent outcome after repeated executions, as per the traditional definition, or if it appears to always pass but contains assertions with a probability of failing above a specified threshold. They evaluated FLASH with 20 open-source machine learning projects written in Python and identified 11 previously unidentified flaky tests, ten of which were confirmed to be flaky by the projects’ developers. The authors further validated their technique by demonstrating that it could detect an additional 11 flaky tests that had been previously identified by developers as evidenced within their subjects’ version control histories.

Pattern Matching

Waterloo et al. [161] performed static analysis on the test code of 12 open-source Java projects. They derived a set of syntactical code patterns associated with common bugs in tests, many of which they believed to be indicative of potential test flakiness. Their analysis aimed to identify instances of tests that matched these patterns, which were split into three families. The first was *inter-test* and contained code patterns regarding the relationships between test cases, in other words, instances of violations of the test independence assumption, such as tests that invoke one another and share static fields or data streams. They cited Zhang et al. [175] to support the value of this family of patterns as an indicator of potential test order dependencies. The second was *external* and referred to tests with dependencies on external resources, specifically, test cases with hard-coded time delays for waiting on asynchronous results, those with unchecked dependencies upon the system state (e.g., reading/modifying environment variables) and tests that assume some network resource will be available during their execution. They specifically highlighted the hard-coded time delay pattern as being previously identified as a common cause of test flakiness, citing

Table 2.9: A comparison of four different studies that applied machine learning to the detection of flaky tests. Static features are those that can be derived without running the test, e.g., number of source lines. Dynamic features require at least one test run, e.g., line coverage.

Study	Classifier	Features
King et al. [82]	Bayesian network [46]	Static and dynamic
Bertolino et al. [159]	k -nearest neighbour [80]	Static only
Pinto et al. [128]	Random forest [143]	Static only
Alshammari et al. [8]	Random forest [143]	Static and dynamic

Luo et al. [105], who also identified network dependency as a possible avenue for flakiness. The third family was *intra-test* and pertained to issues with assertion statements within test cases, such as those that use a serialised version of an entire object, which may contain irrelevant information and thus result in fragile tests that “over check”.

Their results indicated a very low incidence of *inter-test* patterns, which was at odds with what they expected given the prevalence of order-dependent tests as previously identified [175], suggesting that either their patterns were not indicative of such tests or that static analysis alone may be insufficient to identify them. As for the *external* family, the authors found many instances of potentially problematic patterns across their subject set. With regards to hard-coded time delays in particular, indicative of potential *asynchronous wait* flaky tests, their technique identified 807 matches in their subject set. Further manual analysis of 31 such matches showed that they were all true positives, that is, genuine cases of hard-coded time delays but not necessarily flaky tests, leading them to reaffirm the value of their static analysis approach for this particular pattern. Given its association with test flakiness [105], this finding suggests that static analysis may be useful for identifying possible flaky tests, which is particularly valuable given its low cost (which is on the order of minutes when run on a commodity laptop) compared to repeatedly re-executing tests to identify flakiness. Naturally, given its static nature, this approach is unable to verify if its matches indicate genuine, manifest flaky tests, and could thus have poor precision even if it exhibits high recall.

Applying Machine Learning to Detection

Using statically identifiable characteristics and the historical execution data of tests, King et al. [82] showed how to use a Bayesian network to classify a test as flaky. A Bayesian network [46] is a directed acyclic graph in which each node represents a probability distribution conditioned on its antecedents. In this case, test flakiness may be considered a “disease” whose probability is conditional on the observation of a variety of test metrics or “symptoms”, as they described it. They selected a multitude of such metrics, covering characteristics such as complexity, implementation coupling, non-determinism, performance and general stability. Examples of concrete metrics used include the number of assertions in a test, the average execution time, and the rate at which a test alternates between passing and failing based on historical records. The authors evaluated their approach within the context of a software company called Ultimate Software, where they gathered training examples from historical instances of flaky tests being identified, quarantined and eventually fixed. After training and evaluating their model on one of Ultimate Software’s own products, they reported an overall prediction accuracy of 66%.

Bertolino et al. [159] presented FLAST¹, a machine learning model for classifying tests as flaky or not based purely on static features. To represent a test case in their model, they used the *bag of words* technique on the tokens within its source code, such as identifiers and keywords. Bag of words is a common representation in the field of natural language processing, where a sample of text is represented as a typically very sparse vector where each element corresponds to the frequency of a particular token [176]. Their model is a *k-nearest neighbour* classifier [80], which classifies representations of tests cases based on the labels (flaky or non-flaky) of their closest k

¹While their paper was not published until 2021, their tool has been available in prototype form since 2019.

“neighbour” training examples according to some metric over the vector space (which is *cosine distance* in this case) and a classification threshold that determines the proportion of a test’s neighbourhood that would have to be flaky to classify it as such. As training examples, they used the test cases from the subject projects of previous studies that had automatically identified flaky tests [16, 90]. This meant that they did not have to label the test cases as flaky or non-flaky themselves. For each of these projects, they individually evaluated the effectiveness of their model. To that end, they split their set of training examples for each project into 10 equal *folds*. For each fold, they trained their model on the other 9 folds and then evaluated its effectiveness on the remaining fold. Specifically, they calculated the *precision* and the *recall* attained by their model and then calculated the mean of these metrics across the 10 folds. Precision is the ratio of true positives (the number of tests correctly labelled as flaky) over all positives (the number of tests labelled as flaky, correctly or incorrectly). Recall is the ratio of true positives over true positives *and* false negatives (the number of flaky tests, labelled correctly or incorrectly). For each project, the authors performed evaluations of their model under four configurations. These were based on the combination of two values for k and the classification threshold. For k , the authors tried the values of 7 and 3. For the classification threshold, the authors tried 0.95 and 0.5. In the 0.95 case, their k -nearest neighbour classifier would have to find the vast majority of a test’s neighbourhood as flaky to classify it as such, resulting in much more conservative predictions with respect to the positive case. The configuration that offered the best trade-off between precision and recall was $k = 3$ with a threshold of 0.5, giving a mean precision of 0.67 and a mean recall of 0.55, resulting in an *F1 score* of 0.60. F1 score is the harmonic mean of precision and recall and is intended to offer a fair assessment of a model’s accuracy. As a static approach it was very fast, with an average training time of 0.71 seconds and an average prediction time of 1.03 seconds across their set of projects.

Pinto et al. [128] performed a related study, investigating the notion of flaky tests having a “vocabulary” of identifiers and keywords that occur disproportionately in flaky tests and may be indicative of them. To investigate this, they trained five common machine learning classifiers to predict flakiness using “vocabulary-based” features derived from the bodies of test cases. Specifically, their features encoded occurrences of whole identifiers and parts thereof (by splitting each word in a camel case identifier), as well as other metrics such as the number of lines of code in the test case. To train these classifiers, they used flaky tests previously identified by DEFLAKER [16] as positive examples. As negative examples, they repeatedly executed the test suites of the DEFLAKER subject set and selected the same number of tests with consistent outcomes as flaky tests, ensuring they had a balanced dataset. The five types of machine learning classifiers they trained were *random forest*, *decision tree*, *naive Bayes*, *support vector machine*, and *nearest neighbour*. To evaluate these, they split their dataset into 80% for training and 20% for evaluation. Following this, they calculated F1 scores for each classifier. Unlike Bertolino et al., they did not calculate individual scores for each project and then average these scores. Rather, they trained and evaluated each classifier just once, using all the training examples from each project all together. Therefore, it is unknown how the performance of these classifiers varies between projects. Furthermore, projects with more flaky tests, and thus more training examples, would have more impact on the final F1 score than those with fewer flaky tests. Their evaluation reported the F1 scores for each type of classifier, with random forest [143] being the best classifier with a score of 0.95. They also identified the most valuable features for classification in terms of their information gain. Information gain is measured in bits and, in this context, indicates how much information the knowledge of each feature gives towards knowing if a test is flaky or not. The top three features they identified were the occurrences of the tokens `job`, `table` and `id`, suggesting that the presence of these within a test case may be indicative of flakiness.

Studies have attempted to reproduce the findings of Pinto et al. several times in different contexts. Ahmad et al. [4] reproduced their methodology with a set of Python projects, and reported lower precision, recall, and F1 scores for three of the five machine learning classifiers used by Pinto et al. Haben et al. [65] sought to investigate the effectiveness of Pinto et al.’s vocabulary-based feature approach when using a time-sensitive training and evaluation methodology, that is, to train a model with “present” flaky tests and evaluate it on “future” flaky tests, with respect to

some time point or commit. They also evaluated the effectiveness of vocabulary-based features in Python, like Ahmad et al., and went on to consider if identifiers and keywords from the code under test were of any use to flaky test prediction. For their investigation of the time-sensitive methodology, they took six of the 24 projects used in the initial evaluation by Pinto et al., each with at least 30 flaky tests. For each of these six projects, they identified the commit where 80% of the known flaky tests were present, forming the training set, and 20% were yet to be introduced, forming the evaluation set. They then trained and evaluated a random forest classifier individually for each project. Their results showed that, for four of the six projects, the time-sensitive methodology produced poorer results than the “classical” methodology used by Pinto et al.—simply splitting the dataset into 80% for training and 20% for evaluation. Haben et al. argued that the time-sensitive methodology more accurately reflects the expected use case of a flakiness model, since presumably developers would use it to assess test cases as they introduced them. Following this, they adhered to Pinto et al.’s “classical” methodology for nine Python projects. They recorded generally good performance across their Python subject set, with a mean F1 score of 0.80 across each project. From this, they concluded that the vocabulary-based feature approach is generalisable across programming languages. Finally, they investigated whether including features from the code under test when training a vocabulary-based model would improve its performance. Since a vital detail of this approach is that it enables *static* prediction of flaky tests, Haben et al. did not use actual coverage data to identify the code under test associated with each test case. Instead, they used an information retrieval technique to statically estimate which functions of the code under test each test case was likely to cover, from which they extracted identifier counts. Using both the Java and Python subject sets, they found that including features from the code under test did not improve the performance of the model.

Alshammari et al. [8] proposed, implemented and evaluated FLAKEFLAGGER, a machine learning approach for predicting tests that are likely to be flaky. An initial motivating study demonstrated that, across the test suites of 24 open-source Java projects, flaky tests were still being detected after up to 10,000 reruns. This demonstrated the impracticality of straight-forward rerunning as an approach for detecting flaky tests, highlighting the need for alternative methods. Following a literature review, the authors identified 16 test features that they believed to be potentially good indicators of flaky tests. These consisted of eight boolean features regarding the presence or absence of various test smells [12, 50], such as whether or not the test accesses external resources. The remaining features were numeric, such as the number of lines making up the test case, the number of assertions, and the total line coverage of production code by the test. Their approach for collecting these features for a given test was a hybrid of static and dynamic analysis, since not all features (e.g., line coverage) are attainable from static analysis alone. The authors evaluated a variety of machine learning models, finding random forest to be the most effective. As their training and evaluation procedure, Alshammari et al. used stratified-cross-validation with a 90-10 training-testing split [173]. When training a model, they used the SMOTE oversampling technique [23] to ensure a balanced data set with an equal number of flaky and non-flaky training instances. When evaluating a model, however, they did not apply SMOTE in order to reflect the real-life environment in which their model would be applied, where non-flaky tests far outnumber flaky tests. The authors went on to compare their approach to that of Pinto et al. described previously. Alshammari et al. noted that Pinto et al. evaluated their model using a balanced sampling approach, which may have led to an overestimation of their model’s effectiveness. To that end, they evaluated both FLAKEFLAGGER and Pinto et al.’s vocabulary-based feature approach under their training and evaluation methodology, as well as a hybrid approach combining the feature set of FLAKEFLAGGER with vocabulary-based features. Unlike Pinto et al., Alshammari et al. presented their F1 scores for each individual project, and presented an overall average of these, ensuring each project had the same degree of influence on the final score. Their results showed that FLAKEFLAGGER alone had an average F1 score of 0.66, the vocabulary-based approach had an average F1 score of 0.19, and the combination of the two feature sets resulted in an average F1 score of 0.86.

Applying Automatic Tools

Lam et al. [93] set out to investigate the most effective strategy for applying flaky test detection tools. As objects of study, they considered two tools, `iDFlakies` [90] and `NonDex` [61, 140], and as subjects, they reused the *comprehensive* set of the `iDFlakies` study (see Section 2.5.2). The combination of these two tools enabled them to identify flaky tests that were order-dependent via `iDFlakies` (see Section 2.5.2) and implementation-dependent via `NonDex` (see Section 2.5.1), thus covering a wide range of flaky test categories. Initially, they executed these tools on the commits that were sampled as part of the `iDFlakies` subject set to identify flaky tests. For each flaky test, they then ran the tools on the initial commit that introduced it into the test suite. If they found that it was not flaky on this test introducing commit, they then searched for the first commit where the test was flaky, reasoning that this commit would have introduced the flakiness. To that end, they performed a binary search starting with the test introducing commit and the `iDFlakies` commit, eventually converging on the flakiness introducing commit. They identified 684 flaky tests across the `iDFlakies` commits of their subject set, of which they were able to successfully compile and execute the test introducing commits of 245. Of these, 75% were flaky from their test introducing commit, the remaining 25% were associated with a flakiness introducing commit later in their lifetime. Furthermore, they found the median number of commits between the test introducing commits and the flakiness introducing commits of these 25% to be 144, representing 154 days of development time. This led them to suggest that running flaky test detectors immediately after tests are introduced and then periodically every 150 commits or so would achieve a good detection-to-cost ratio, as opposed to running them after every commit, which would be prohibitively expensive for many projects.

Conclusion for RQ3.1: What techniques and insights are available for detecting flaky tests in general? The most straight forward techniques for automatically detecting flaky tests are based on repeatedly executing them [16, 92]. Since this can be very time consuming, more efficient and more targeted approaches have been proposed. One technique makes use of the difference in coverage between consecutive versions of a piece of software to identify flaky tests as those whose outcome changes despite not covering any modified code. An evaluation of this approach found that it was able to identify 96% of the flaky tests identified by repeated test suite runs [16]. One paper evaluated a tool that randomises the implementations of various Java classes which have non-deterministic specifications with the aim of manifesting implementation-dependent flaky tests [140]. Across 21 open-source Java projects, this technique identified 60 flaky tests. Another study presented an approach targeting flaky tests of the *asynchronous wait* and *concurrency* categories by introducing CPU and memory stress to impact thread timings and interleaving during test suite reruns [144]. An empirical evaluation found that this approach could detect such flaky test at a faster rate than standard rerunning alone. One paper presented an approach for detecting flaky tests of the *randomness* category, specifically within machine learning projects [35], and demonstrated its effectiveness by detecting 11 previously unknown flaky tests. Techniques from the field of machine learning have been applied to flaky test detection, with several studies considering the presence of particular identifiers in test code, and other general test characteristics, as potential predictors of flakiness [8, 128, 159]. With regards to prediction based on identifiers, also known as the vocabulary-based model [65], three separate studies presented different degrees of effectiveness on the same subject set [8, 65, 128], highlighting the impact of different evaluation methodologies.

2.5.2 Detecting Order-Dependent Tests

Given their particular costs to methods for test suite acceleration, a thread of work has emerged specifically concerned with detecting order-dependent tests. Some authors has proposed methods that, while not detecting order-dependent tests directly, may still be of use to developers for debugging them [62, 74]. For detecting order-dependent tests directly, one early study [113] explained

how isolating test cases can identify bugs that are masked by implicit dependencies between other test cases. Since then, more sophisticated and multi-faceted approaches have emerged, often motivated by the desire to mitigate against the unsoundness that order-dependent tests impose upon techniques for test suite prioritisation [175] and parallelisation [15, 17]. Many of these approaches directly build upon one another, iteratively making improvements or adding additional features [15, 17, 47, 175]. Three of these studies perform evaluations with the developer-written test suites of the same four Java projects [15, 47, 175], the results and analysis costs of which are summarised in Table 2.10. Other work has presented techniques based on repeating test suites in different orders in a systematic way as a method for identifying order-dependent tests [90, 163].

Brittle Assertions

Huo et al. [74] developed a way to detect *brittle assertions*, assertion statements that are affected by values derived from inputs uncontrolled by the test, and *unused inputs*, inputs that are controlled by the test but that do not affect any assertions. Brittle assertions in particular may create an opportunity for order-dependent tests to arise since they make the outcome of a test depend upon inputs that it does not control, but that could potentially be set by another test. To detect brittle assertions, they presented a technique based on input tainting. This involves associating “taint marks” to uncontrolled inputs that are propagated across data and control dependencies to identify if they eventually end up affecting an assertion statement. The technique, targeting Java programs, selects the static and non-final fields of a test’s containing class as the initial uncontrolled inputs. For example, when a test assumes that a particular field is at its default value and makes use of it in an assertion statement, this is an uncontrolled input since the test does not set the default value and another source (i.e., a previously executed test) could have modified it. In an attempt to eliminate false positives, for each input identified as the cause of a brittle assertion, their technique re-executes the respective test and mutates the value of the uncontrolled input. In the case where this does not impact the test outcome, the result is considered a false positive. The authors implemented their approach as the ORACLEPOLISH tool, evaluating it with a subject set of over 13,609 tests, within which 164 brittle assertions were detected and verified as true positives. Their approach incurred a run-time cost of between five to 30 times that of a regular test suite run.

Test Pollution

Gyori et al. [62] proposed a technique that specifically identifies tests that leave side-effects, as opposed to the tests that are impacted by them. Such tests may induce order dependency in subsequently executed tests, and thus their detection may assist developers in debugging them. Their approach models the internal memory heap as a multi-rooted graph, with objects, classes and primitive values as nodes and fields as edges. The roots of this graph represent global variables accessible across test runs, that is, the static fields of all the loaded classes in the current execution. Their approach compares the state of the heap graph before the setup phase and after the teardown phase of a test method’s execution to identify any changes, or as they termed them, “state pollution”. They implemented their technique as a tool named POLDET, which integrates with the popular Java testing framework JUNIT [248], complete with various measures for ignoring side-effects upon irrelevant global state and thus avoiding false positives, by ignoring mock classes for example. They leveraged a modified JUNIT test runner to invoke their state graph building logic, which is implemented using reflection upon all loaded classes at each “capture point”, i.e., before a test’s setup method is invoked and after its teardown method. Furthermore, they equipped POLDET with functionality for capturing the state of the file system across test runs to identify file system pollution, another potential avenue for test order dependencies.

The authors evaluated their tool with the test suites of 26 open-source projects, in total comprising over 6,105 test methods. They found that, when it was configured to ignore irrelevant state, POLDET identified 324 heap polluting tests and a further 8 that polluted the file system. To determine if a positive result for heap pollution was true or false, the authors manually in-

Table 2.10: The number of developer-written order-dependent tests detected by DTDETECTOR [175], ELECTRICTEST [15] (ETEST), and PRADET [47] (PDT) and the time taken to do so in seconds. The figures for analysis cost are not directly comparable between studies, since they used machines of different specifications. For DTDETECTOR, figures reported are those of Zhang et al. [175]. For ELECTRICTEST, the number of order-dependent tests reported are the numbers of tests identified as reading a shared resource previously written to by another test. Since this does not necessarily imply that they are order-dependent, some may be false positives. An asterisk (*) indicates that the execution timed out after 24 hours.

DTDETECTOR [175]											
Subject	Tests	Randomised				Exhaustive				Dep. Aware	
		Rev.	$n = 10$	$n = 100$	$n = 1000$	$k = 1$	$k = 2$	$k = 1$	$k = 2$	ETEST [15]	PDT [47]
Order-Dependent Tests Detected											
Joda-Time	3875	2	1	1	6	2	2*	2	2 *	121	8
XML Security	108	0	1	4	4	4	4	4	4	103	4
Crystal	75	18	18	18	18	17	18	17	18	39	2
Synoptic	118	1	1	1	1	0	1	0	1	117	4
Total	4176	21	21	24	29	23	24	23	25	380	18
Time Taken in Seconds											
Joda-Time	3875	11	57	528	5538	1265	86400*	291	86400*	2122	46
XML Security	108	11	65	594	5977	106	11927	93	3322	57	146
Crystal	75	2	14	131	1304	166	7323	95	4155	22	106
Synoptic	118	1	7	67	760	25	3372	24	1797	34	14914
Total	4176	26	143	1320	13579	1562	109022	503	95674	2235	15212

spected each one, labelling them as a true positive if they could write another test whose outcome would depend on whether it was run before or after the reported polluting test (thus, by definition, creating an order-dependent test), or a false positive if they could not. They identified 60% of the positive results as true this way, suggesting that their tool may have some issues with its precision. In terms of efficiency, the authors found that POLDET had an overhead of 4.5 times the usual test run duration, but with significant variance across different projects.

Dependence Aware

Zhang et al. [175] proposed four algorithms for manifesting order-dependent flaky tests in Java test suites by comparing their outcomes when executed as normal to when executed in a different order. The first, *reversal*, simply reverses the test run order. The second, *randomised*, shuffles the test run order. The third, *exhaustive*, executes every k -permutation of the test suite in isolation, that is, in a separate Java Virtual Machine. The fourth, *dependence-aware*, aims to improve the efficiency of the *exhaustive* technique by filtering permutations that are unlikely to reveal a test order dependency, which it does by analysing the access patterns of *shared resources*, such as global variables and files, across test runs. In the case where $k = 1$, the dependence-aware algorithm performs a test run in the default order to establish their baseline outcomes. Any tests that do not read or write any shared resources during this run are considered unlikely to be involved in a test order dependency and are filtered out. The remaining tests are executed in isolation and if their outcomes differ from the baseline then they are reported as order-dependent. In the case where $k \geq 2$, each test is first executed in isolation to establish the baseline, again monitoring reads and writes to shared resources. When generating permutations, if it is the case that each test does not read any shared resources that are written to by any previous tests, as measured during the baseline isolation run, then the permutation is discarded. To record reads and writes to global variables, their approach uses bytecode instrumentation to monitor accesses of static fields, conservatively considering any read to also be a potential write. To identify test order

dependencies facilitated by files, DTDETECTOR installs a custom `SecurityManager` [274] that monitors the files read from and written to by each test.

Zhang et al. evaluated each of these approaches on the developer-written and automatically generated test suites of four open-source projects implemented in the Java programming language. They found, in both cases, that the *reversal* technique manifested the fewest order-dependent tests but was also the cheapest in terms of run-time by a considerable margin. The *randomised* approach with 1,000 repeats manifested the most, but had a relatively significant cost, proving to be three orders of magnitude greater than *reversal*. The *exhaustive* and *dependence-aware* techniques with $k = 1$ had run times comparable to *randomised* with 100 repeats but manifested order-dependent tests. With $k = 2$, both techniques timed-out after 24 hours and failed to perform better than the *randomised* approach. These results indicate that executing a test suite in reverse may be an acceptable baseline approach, since it could detect 72% of all identified order-dependent tests and ought to take no longer than a standard test suite run.

Object Tagging

Bell et al. [15] aimed to develop a technique more efficient than DTDETECTOR [175] for identifying order-dependent tests. Their approach differentiates between two types of dependency relationship. The first, *read-after-write*, refers to the case where `testB` reads a shared resource last written to by `testA`, such that `testB` must be executed after `testA` to preserve the dependency. The second, *write-after-read*, describes the scenario where `testC` also writes to the same resource as `testA`, meaning that `testC` should not be executed between `testA` and `testB`. They implemented their technique as a tool, called `ELECTRICTEST`, that can identify instances of the two relationships during an instrumented test suite run. The authors explained that recording accesses to static fields would be insufficient to accurately detect all test order dependencies facilitated by in-memory shared resources. This is because a test could *read* a static field to get a pointer to some object and then *write* to that object's instance fields, with the whole operation being reported as just a read. While DTDETECTOR gets around this problem by conservatively considering all static field reads to also be potential writes, `ELECTRICTEST` takes a more fine-grained approach. To detect *read-after-write* relationships, it forces a garbage collection pass after each test run and tags any reachable objects as having been written to by the test that was just executed, provided that they have not already been tagged as such by another test. In subsequent test executions, `ELECTRICTEST` is notified when a tagged object is read from, in which case the reading test is marked as being dependent on the writing test. The tool follows a similar approach for identifying cases of *write-after-read*. This tagging functionality is provided by the *JVM Tooling Interface* [244]. As well as in-memory resources, `ELECTRICTEST` uses the Java Virtual Machine's built-in `IOTrace` features to detect test order dependencies over external resources such as files.

Evaluating their tool against DTDETECTOR [175], using the same four developer-written test suites, they found that `ELECTRICTEST` could detect all the same order-dependent tests as DTDETECTOR and many more, indicating that the recall of `ELECTRICTEST` was at least as good as that of DTDETECTOR. Since `ELECTRICTEST` requires only a single test suite run, it was found to be up to 310 times faster than the *dependence-aware* mode of DTDETECTOR. The fact that `ELECTRICTEST` does not verify the dependencies it detects by executing the concerned tests means that its precision may be poorer since, as Bell et al. noted, the order-dependent tests it identifies may not be *manifest*, in other words, while they may read resources written to by previous tests, their outcomes may not be impacted. One could consider these cases as false positives since they do not hinder test suite acceleration techniques (see Section 2.4.2), one of the primary motivations for detecting order-dependent tests. This is a particularly pertinent point given the cost of accommodating order-dependent tests (see Section 2.6.1), which means that too many false positives could become a significant burden. A further evaluation of `ELECTRICTEST` with the test suites of ten open-source projects not previously used, with an average of 4,069 test methods between them, indicated a mean relative slowdown of 20 times a regular test suite run and an average of 1,720 test methods that wrote to a shared resource that was later read from and 2,609 that read a previously written resource.

Dependency Validation

Building on the work of ELECTRICTEST [15], Gambi et al. [47] presented PRADET. One limitation of ELECTRICTEST, as previously explained, is that it may identify order-dependent tests that are not *manifest*, meaning that breaking their dependencies by reordering the test suite does not result in a different test outcome. Initially, PRADET operates in a similar way to ELECTRICTEST, monitoring access patterns of in-memory objects between test executions to identify instances of possible test order dependencies. One improvement of PRADET over ELECTRICTEST at this stage, as the authors explained, is in its handling of `String` objects and enumerations, respecting their pooled and immutable implementation, which results in fewer false positives.

Modelling a test suite as a graph, with tests as nodes and possible dependencies as directed edges, PRADET verifies the dependencies it identifies by selecting edges, inverting them as to break the dependency, and generating a corresponding test run schedule using a topological sort on the graph. For efficiency, the schedule does not contain tests that are irrelevant to the selected dependency, that is to say, those that do not belong to the weakly connected component. When executing the schedule, in the case where the dependent test corresponding to the inverted edge does not produce a different outcome, as compared to a regular test suite run, the dependency is considered non-manifest and is removed. The tool follows a *source-first* strategy, selecting edges corresponding to the later executed tests first. Compared to a random approach, this allows initially impossible schedules, as in the case where inverting a dependency leads to a cycle, to be reliably deferred to a later stage, such as when the cycle is broken by another candidate dependency being removed. Figure 2.6 gives a visual summary of this verification approach used by tools like PRADET.

They performed two evaluations of their tool, comparing PRADET to DTDETECTOR using its *reverse* mode and its *exhaustive* mode with $k = 1$ and $k = 2$. The first evaluation used the developer-written test suites of the same four subjects as used in its initial evaluation of DTDETECTOR by Zhang et al. [175]. Their findings suggested that PRADET identified more order-dependent tests than any of these modes, although this is at odds with what Zhang et al. reported (and as presented in Table 2.10), which would indicate that PRADET actually detected the fewest. The second evaluation used a wider subject set of 15 projects. In this setup, the *exhaustive* mode with $k = 2$ detected the most order-dependent tests by a significant margin, followed by the *reverse* mode, and then by PRADET. In terms of analysis cost, PRADET was more than five times faster than the *exhaustive* mode with $k = 2$, though it was about ten times slower than with $k = 1$. Unsurprisingly, the *reverse* mode was the fastest, since it only requires a single test suite run with no isolation overhead.

Web Applications

Biagiola et al. [17] presented an approach, implemented as a Java program named TEDD, for building dependency graphs of tests within the test suites of web applications. In this context, a dependency graph consists of nodes representing individual SELENIUM test cases and directed edges representing test order dependencies. Previous approaches based upon read/write operations on Java objects [15, 47, 175] are unsuitable in this domain since web applications are more prone to test order dependencies from persistent data stored on the server-side and implicit shared data via the Document Object Model on the client-side. Their multi-faceted technique consisted of four stages.

In the first stage, the technique extracts the initial dependency graph by iterating through each test case in the test suite's original order and identifying the set of named inputs submitted by each test, such as the values inserted into input fields of a web page via `sendKeys` [249]. Then, for every following test, the set of inputs *used* when evaluating oracles are also identified. When these two sets have an intersection, and at least one test submits an input that another uses, a candidate dependency is added to the dependency graph between the two tests in question. They referred to this technique as *sub-use string analysis graph extraction*. Alternatively, the tool can connect every pairwise combination of tests according to the original test run order, a baseline

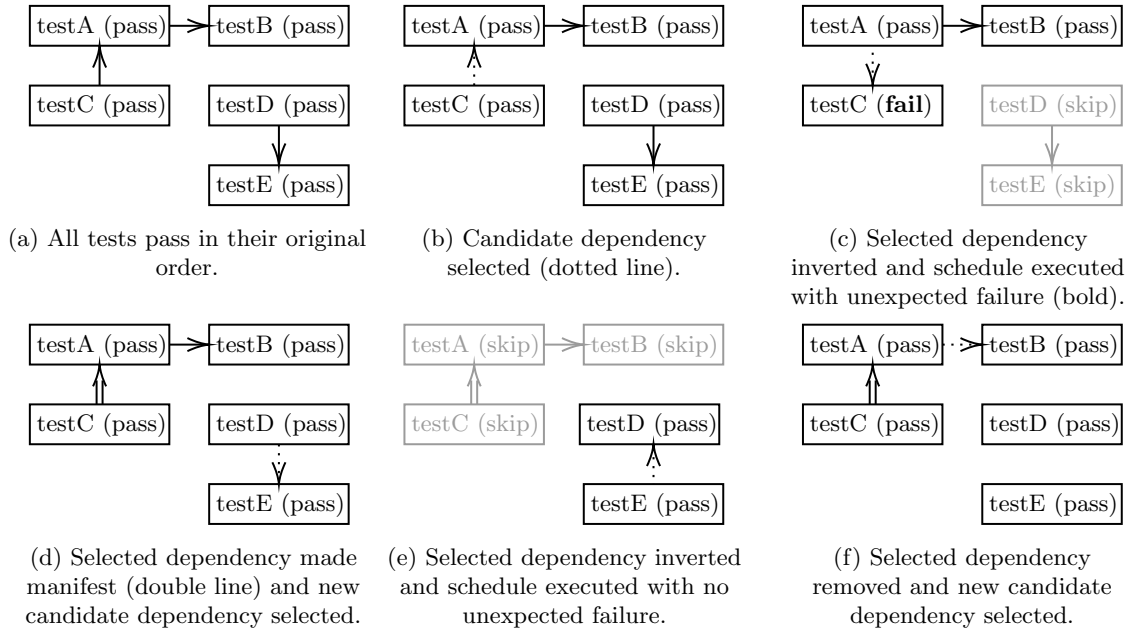


Figure 2.6: An illustration of several iterations of the dependency verification stage used by PRADeT [47] and TEDD [17]. Test nodes are represented by rectangles and dependency edges by arrows. The tests of the disconnected components with respect to the inverted dependencies in parts (c) and (e) are skipped, since they are deemed irrelevant for verifying the dependency.

method they referred to as *original order graph extraction*.

In the second stage, natural language processing techniques applied to the names of test cases are used to filter likely false dependencies. A technique known as *part-of-speech-tagging* [85] is used to identify the verbs and nouns of each name. Semantic analysis on each verb decides the CRUD operation (i.e., *create*, *read*, *update*, or *delete*) to which it is closest. This is considered with respect to the nouns in the name to identify if they suggest dependence. For instance, the names `updateFile` and `readFile` would pass this filtering stage since they both concern the same noun, `File`, and the verbs suggest a read-after-write dependence. Their approach offers three configurations for this filtering stage: consider only the verb in the name, consider the verb and the direct object it applies to only, or the most comprehensive, consider the verb and all the nouns in the name of the test case.

In the third graph-building stage, the TEDD tool validates the candidate dependencies that survived the filtering stage using a similar inversion approach to that of PRADeT [47]. Given that the filtering stage may not be conservative, TEDD attempts to recover any dependencies that may be missing from the graph. To that end, after executing a schedule with a candidate dependency inverted and witnessing a failure in the corresponding dependent test, when the test was otherwise passing in the original order, TEDD then generates and executes another schedule with the dependency *not* inverted. This is to identify if the failure was due to the inverted dependency or a potentially missing one. In the case were one or more tests unexpectedly fail in the non-inverted schedule, it is assumed that one or more dependencies are missing and need to be recovered. To do so, TEDD connects the first failing test to each of its preceding tests in the original test run order, that were not executed in the non-inverted schedule, as candidate dependencies. In turn, these newly added edges are inverted and tested as the validation stage continues until all candidate dependencies from the previous stages have either been verified or removed.

In the fourth and final stage, missing dependencies are recovered for disconnected components, these are nodes (tests) with no outgoing edges (no dependents) or those that are fully isolated with no incoming edges (no dependents) either. To that end, all such tests are executed in isolation

in separate Docker containers [214]. For each failing test, TEDD connects it as a candidate dependent to every preceding test in the original order. For each passing test that is not an isolated node, having a non-zero in-degree, TEDD executes every schedule containing the test according to the dependency graph's current state. If a test in any of these schedules fails, TEDD connects it with every preceding test as before. To verify these newly added candidate dependencies, it re-executes the third graph building stage.

The authors went on to evaluate TEDD using six open-source subjects. They considered every combination of both the original order and the sub-use string analysis graph extraction methods along with four initial filtering modes, no filtering plus the three granularities of natural language processing techniques, for a total of eight (2×4) overall configurations. In terms of effectiveness, they found that every configuration found roughly the same number of order-dependent tests, between 32 and 34. In terms of performance, they found that the combination of sub-use string analysis with filtering considering every noun to be the fastest. Analysing filtering methods specifically, in the case of original order extraction, the verb only, direct object only and all nouns filtering methods achieved run time savings of 27%, 67% and 70% respectively over no filtering. For sub-use string analysis, these were -69% (i.e., in this case the filtering increased the total run time), 17% and 28%, respectively. These findings indicate that the most comprehensive natural language filtering technique of examining the verb and all the nouns in the names of tests was the most effective.

Repeating Failing Orders

Lam et al. [90] presented IDFLAKIES, a framework and tool for detecting flaky tests in Java projects. This framework can classify a flaky test as being the result of a test order dependency or not. Their approach involves repeatedly rerunning the test suite in a modified order, the type and granularity of which can be specified by the user, that is, reversed or randomised at the class-level, method-level or both. Initially, the test suite is repeatedly executed to determine which tests pass in their original order. Following this stage, the test suite is repeatedly executed in a modified order. Upon a new test failure, the test suite is re-executed up to and including the failing test both in the modified order that witnessed the failure and in the original test suite order where the test was previously passing consistently. When the test fails in the modified failing order but passes in the original order, it is classified as an order-dependent (OD) flaky test, otherwise, it is classified as a flaky test that is not order-dependent (NOD).

They took projects from previous work [16] and augmented them with 150 popular projects from GitHub, to arrive at a total of 2,921 modules across 183 projects, which they referred to as their *comprehensive* subject set. They took a further 500 projects from GitHub, disjoint from their *comprehensive* set, to form what they called their *extended* subject set. Using the union of these two subject sets, they evaluated their tool in a configuration that randomises the test run order at both the class and method granularity (i.e., shuffles the order of test classes and then shuffles the order of test methods within these classes). They identified a total of 213 OD flaky tests and 209 NOD flaky tests across 111 modules of 82 projects. This would suggest that just over 50% of flaky tests were order-dependent, which is at odds with the findings of Table 2.5. This discrepancy could be due to the existence of many more order-dependent tests than developers are aware of, which makes sense if they are not using automatic tools such as *iDFlakies*. Using only the *comprehensive* set, they then compared the effectiveness of the different configurations of *iDFlakies* at detecting flaky tests. They found that randomizing at both the class and method level, as before, was the clear winner, manifesting 162 OD tests and 74 NOD tests. For comparison, the runner up, randomizing at the class level only, identified 40 OD tests and 53 NOD tests.

Later findings by Lam et al. [92] suggest that classifying tests into OD and NOD may be too coarse. Using a subset of the modules used to evaluate IDFLAKIES, they calculated the failure rates (i.e., the ratio of failures to total runs) of 96 flaky tests identified by repeatedly executing test suites in different test class orders. A chi-squared test of independence identified that 70 such flaky tests had failure rates that were different across orders to a statistically significant degree. They concluded that such tests failed more often in some orders and less often in others and were

therefore likely to be non-deterministically order-dependent (NDOD), in other words, somewhere between OD and NOD.

Tuscan Squares

Wei et al. [163] proposed a technique for generating test run orders that are more likely to manifest order-dependent tests than simply sampling orders at random (such as the randomised mode of IDFLAKIES [175]). Given that the majority of test-order dependencies depend on only a single other test [141, 175], their technique generates the provably smallest number of test run orders in some instances, such that every pair of tests is covered, that is, executed consecutively both ways. For example, for 4 test cases, each of the 12 permutations of length 2 can be covered by the following 4 test run orders: $\{\langle t_1, t_4, t_2, t_3 \rangle, \langle t_2, t_1, t_3, t_4 \rangle, \langle t_3, t_2, t_4, t_1 \rangle, \langle t_4, t_3, t_1, t_2 \rangle\}$. To determine these orders, their technique computes the *Tuscan square* [53] from the field of combinatorics. A Tuscan square for the natural number m is equivalent to a decomposition of the complete graph of m vertices into m Hamiltonian paths. This object contains m rows, each of which is a permutation of the integers $[1..m]$. For every pair of distinct numbers in that same range, there exists a row in the Tuscan square where they occur consecutively. In this context, each number represents a test case and each row represents a test run order.

Having observed that Java test suite runners do not interleave test cases within classes, the authors extended the concept of test pairs to that of *intra-class* and *inter-class* pairs. For a set of test run orders to attain full intra-class pair coverage, for every test class, each pair of test cases must be executed consecutively. For a Java class with n tests, there are $n(n-1)$ intra-class pairs. To attain full inter-class coverage, for every pair of classes, each test case from the first class must be executed consecutively with each test case from the second class. For a test suite with k test classes, there are $2 \sum_{1 \leq i < j \leq k} n_i n_j$ inter-class pairs. In order to avoid generating test run orders that test runners would not execute (due to the interleaving of test classes), and thus potentially detecting false-positive order-dependent tests, the authors devised a randomised algorithm, based on computing Tuscan squares, to ensure all intra-class and inter-class pairs are covered with substantially less cost. They evaluated their algorithm on 121 modules from the same set of Java projects used to evaluate IDFLAKIES [90], finding that it only required 51.8% of the test case runs that executing every pair of tests exhaustively in isolation would require, a trivial method of ensuring that all test pairs are covered.

Conclusion for RQ3.2: What methods have been developed to detect order-dependent tests?

One early study presented a technique for detecting brittle assertions, which the authors reasoned may indirectly detect order-dependent tests [74]. After implementing their technique as a Java tool named ORACLEPOLISH, they identified 164 brittle assertions across a subject set of 13,609 test cases. A later study presented POLDET, a tool for detecting tests that modify shared program or environmental state, thus potentially creating test-order dependencies. One issue with their approach was that it was difficult to identify if the tests it detected would genuinely go on to induce order dependency in other tests. With the aim of detecting order-dependent tests directly, another study presented DTDETECTOR, a multi-faceted approach consisting of four modes of varying complexity for detecting order-dependent tests [175]. Following an evaluation upon four open-source Java projects, the authors found that the more complex modes incurred a very high run-time cost and did not perform significantly better than the simpler ones. Following this, two tools emerged that improved upon some of DTDETECTORS short-comings, these were ELECTRICTEST and PRADET [15, 47]. Unfortunately, the former of these was prone to false positives and the latter was shown to incur a potentially prohibitive run-time cost. Focusing specifically on web applications, a later study presented TEDD, a multi-stage technique that included the novel application of natural language processing techniques on the names of test cases. Following an evaluation on six open-source projects, the tool was able to identify between 32 and 34 flaky tests depending on its configuration. Based on

executing test suites in randomised orders, another study presented IDFLAKIES, a framework for detecting flaky tests and classifying them as order-dependent or not [90]. Following a large empirical evaluation upon 111 modules of open-source Java projects, the authors identified 422 flaky tests, just over half of which were order-dependent. Finally, a later study presented a more systematic approach to detecting order-dependent tests via rerunning the test suite in different orders, based on the mathematical concept of Tuscan squares [163]. Additionally, the technique was mindful of test run orders that typical test runners would not execute.

2.6 Mitigation and Repair

Having examined techniques for detecting flaky tests, I now turn to approaches for their mitigation and repair. The mitigation strategies I examine attempt to limit the negative impacts of flaky tests without explicitly removing or repairing them. Once again, given their specific costs to test suite acceleration, as explained in Section 2.4.2, I consult several studies on the mitigation of order-dependent flaky tests. Beyond that, I examine sources on the mitigation of flaky tests with regards to some of the more specific testing-related activities previously identified as being negatively influenced by test flakiness. In the context of repairing flakiness, I present and analyze studies offering advice on repairing specific categories of flaky tests, listed in Table 2.4, derived from previous fixes. I then go on to examine techniques that may assist developers in fixing flaky tests and those capable of repairing them automatically [141, 175]. To answer **RQ4**, I address two sub-research questions, with the findings summarised in Table 2.11.






RQ4.1: What methods and insights are available for mitigating against flaky tests?


This addresses techniques that minimise the negative impacts of flaky tests without explicitly removing or repairing them. My answer considers adjustments to the methodologies of many of the testing activities identified as being negatively impacted by flaky tests. (see Section 2.6.1.)

RQ4.2: What methods and insights are available for repairing flaky tests? By answering this question I provide insights into how developers may eliminate the flakiness in their test suites. The answer presents common strategies employed by developers and discusses which pieces of information are the most important for repair. I also examine a technique for the automatic repair of order-dependent tests. (see Section 2.6.2.)

Table 2.11: Summary of the findings and implications answering **RQ4**: What insights and techniques can be applied to mitigate or repair flaky tests? Relevant to researchers (🎓) and developers (</>).

Finding	Implications	Source
</> Isolating the execution of tests in their own processes is an expensive mitigation for order-dependent tests with an average time overhead of 618%. By only reinitializing the relevant program state between tests runs, as with VMVM, the same effect can be achieved with an average overhead of 34%.	While full process isolation for each test run ought to eliminate all test order dependencies facilitated by shared in-memory resources, and is straightforward to implement, developers should avoid using it due to its prohibitive costs .	[14]
🎓 </> When order-dependent tests are present and known, scheduling approaches for sound test suite parallelisation achieved an average speedup of 7 times a non-parallelised test suite run. This is compared to an average speedup of 19 times if order-dependent tests did not have to be considered.	It is possible to achieve sound parallelisation in test suites with known order-dependent tests, placing further value upon tools for detecting them (see Section 2.5.2) beyond highlighting their presence. Yet, test suite parallelisation is most effective when no order dependent tests are present , so repairing the underlying test order dependencies ultimately may be necessary.	[15]

</> When order-dependent tests are present and not known, certain configurations for test suite parallelisation are more accommodating than others. An evaluation found the most sound to witness an average of 2% of tests failing with a speedup of 1.9 times a non-parallelised test suite run. The least sound saw an average of 24% of tests failing and a speedup of 12.7 times.	When order-dependent tests are not known in advance, reliability may become a trade off for speed when applying test suite parallelisation. Given the poor speedup of the most sound configuration, it may be preferable to apply tools to detect order-dependent tests so they can be specifically mitigated, or better, repaired. [22]
 </> An algorithm for re-satisfying known test order dependencies broken by the application of test suite prioritisation, selection or parallelisation reduced the number of failed order-dependent tests in developer-written test suites by between 79% and 100%.	Beyond test suite parallelisation, detecting order-dependent tests is beneficial to the sound application of other test suite acceleration techniques such as prioritisation and selection. [91]
</> One study demonstrated that a technique for sorting test failures by their stability in the context of GUI regression testing was able to rank the failures witnessing genuine, known bugs over those that were more flaky.	Given the specific factors that lead to flakiness in GUI testing (see Section 2.3.3), developers may find it useful to apply tools that specifically mitigate against them. [49]
 Repeating and isolating test executions in the mutant killing run of mutation testing was shown by one experiment to reduce the number of mutants with an unknown killed status by 79%.	Those using mutation testing to assess the quality of their test suites should consider how repeating and isolating tests may improve this method's reliability. [141]
 Several modifications to EVOSUITE, a test suite generation tool, reduced the number of generated flaky test methods from an average of 2.43 per class to 0.02.	Given the capability of automatic test generation tools to quickly generate many tests, it is important to develop methods for decreasing the potentially significant amount of test flakiness often introduced into automatically created test suites. [10]
</> Between 71% to 88% of historical repairs of flaky tests were exclusively applied to test code .	By fixing a flaky test, a developer may identify weakness in the code under test , as evidenced by the fact that some repairs of flaky tests pertained to source code outside of the test suite. [37, 89, 105]
</> Between 57% and 86% of previous fixes of <i>asynchronous wait</i> flaky tests and up to 46% of <i>concurrency</i> flaky tests involved the addition or modification of an explicit waiting mechanism such as <code>waitFor</code> .	Developers should prefer waiting mechanisms such as <code>waitFor</code> over fixed time delays such as via the <code>sleep</code> method, which require a developer to estimate timings that may be inconsistent across machines. [37, 105]
 The nature, the origin, and the context leading to the failure , of a flaky test were rated by developers as the most important information needed to repair it.	Automatic approaches for assisting developers in repairing flaky tests should focus on retrieving these pieces of information as they would likely be the most useful. [37]
 An approach for identifying the code locations that likely contained the root cause of a flaky test, based upon comparing passing and failing execution traces, was considered useful for repairing them in 68% of cases.	Researchers should consider how techniques for automatically identifying the causes of flaky tests could be useful for developers. [178]

 Techniques have emerged for the automatic repair of order-dependent flaky tests and implement-dependent flaky tests , with generated fixes submitted to the repositories of open-source projects and accepted by their developers.	Initial techniques for the automatic repair of flaky tests show promising results but further work is required to address other categories of flakiness . [141, 174]
---	---

2.6.1 Mitigation

As well as detecting flaky tests, many sources have proposed and evaluated techniques for the mitigation of their negative impacts. Many of these studies specifically examine order-dependent tests with respect to minimizing their costs to test suite acceleration techniques, given the well documented issues that they cause. One such source presented a technique for *unit test virtualisation*, a faster alternate to isolating each test case execution in its own process, a technique shown to be particularly costly in terms of execution time [14]. Other studies have proposed revisions and alternatives to various approaches for test suite prioritisation, selection and parallelisation in the face of order-dependent tests [15, 22, 91]. As well as test order dependencies, other techniques have been examined for reducing the costs of flaky tests upon user interface testing [48, 49], mutation testing [139] and automatic test suite generation [10].

Order-Dependent Tests

Bell et al. [14] presented a lightweight approach for mitigating against order-dependent tests. Initially, they examined the prevalence and the overhead of executing test cases in isolation from one another, by executing each one within their own process. This strategy is used to prevent any state-based side effects of each test case run from impacting later tests, thereby eliminating test order dependencies facilitated by shared in-memory resources. By parsing the build scripts of over 591 open-source Java projects, they found that 240 of them executed their tests with isolation, suggesting that the practice was commonplace. Their results also indicated that there was a positive correlation between the probability that a project used isolated test runs and both its number of tests and its lines of code. This suggests that the developers of more complex projects were more likely to have experienced order-dependent tests. They performed an evaluation with 20 open-source Java projects, where they executed each of their test suites with and without isolation to calculate the time cost of this strategy. They found that, on average, test runs using isolation had a time overhead of 618%, indicating that it is a very expensive mitigation². Having established the significant cost of per-test process isolation, they proposed a lightweight alternative, which they implemented as a tool named VMVM for Java projects. Their high-level approach is to identify which static fields of classes could act a vector for state-based side effects. The classes containing such fields are then dynamically reinitialised, as opposed to reinitializing the entire program state between each test run. Initially, static fields are labelled as “safe” if they hold constant values not derived from other non-constant sources, as identified via static analysis of the source code. Under normal circumstances, the Java Virtual Machine initialises a class (if not yet done so) when it is instantiated, when one of its static methods or fields is accessed or when explicitly requested to do so via reflection. Through bytecode instrumentation, VMVM ensures that all classes containing unsafe static fields are reinitialised under those same conditions repeatedly, thus eliminating any possible side effects that they may propagate between test case executions. An empirical evaluation found that their approach was significantly more efficient than full process isolation, with an average time overhead of only 34%. Furthermore, their results indicated that, under VMVM, test outcomes were the same as when executed with isolation, indicating that their more efficient approach was also just as effective for its intended purpose.

²In recent years, Nie et al. [117], including Bell, identified and submitted a patch for a bug in Maven [253], considerably reducing the overhead of isolation.

As part of the evaluation of their order-dependent test detection tool, ELECTRICTEST, Bell et al. [15] proposed two approaches for test suite parallelisation that respect test order dependencies. Given a test dependency graph, generated by their tool or a similar one such as PRADET [47], they described a *naive* scheduler and a more optimised *greedy* one that both group tests into units that can be safely executed in parallel. The naive scheduler simply traverses the graph and groups tests into chains representing dependencies, where each test appears in only one group containing all the dependencies for every test within it. The greedy scheduler takes into account the execution time of each test as well as their dependency relationships to opportunistically achieve better parallelisation while still respecting order dependencies, by allowing some tests to appear in multiple groups. For example, given ten CPUs and ten tests which all take ten minutes to run, and are all dependent on a single test that takes 30 seconds to run, the naive scheduler would place each of these tests into a single execution group where the 30 second test would run first, achieving no parallelisation. In contrast, the greedy scheduler would create ten groups, each containing the 30 second test followed by one of the ten minute tests. While this duplicates the execution of the faster test ten times, the speedup attained from the parallelisation of the ten slower tests, which is now possible given their satisfied dependency, far outweighs the cost. Using ELECTRICTEST to generate the dependency graphs, they evaluated their two schedulers with ten open-source Java projects using a 32 CPU machine. They found that their naive scheduler attained an average speedup of 5 times a non-parallelised test suite run, whereas their greedy scheduler achieved an average speedup of 7 times. For comparison, they measured the speedup they would have achieved had they not had to respect any test order dependencies and found this to be 19 times on average. These findings demonstrate that order-dependent tests can be mitigated to achieve sound test suite parallelisation (i.e., not leading to inconsistent outcomes) but at the cost of considerable effectiveness.

Candido et al. [22] offered several insights into applying test suite parallelisation to Java test suites with order-dependent tests not necessarily known in advance. They found that forking the Java Virtual Machine process to achieve parallelisation and allocating each instance a portion of the test classes to execute in serial was the most robust approach with respect to manifesting the fewest test failures, in tests that were otherwise passing when executed in serial. Under this strategy they observed approximately 2% of tests failing on average within the test suites of 15 open-source projects. This method was also the least effective in terms of speeding up the test run, with Candido et al. reporting an average speedup of only 1.9 times a serial test suite run. This was as opposed to more flexible configurations that parallelised using multiple threads within a single process and executed test methods concurrently, which were found to be considerably unsound. One such strategy resulted in an average of up to 24% of failed tests in the most extreme case. However, this was also the most effective configuration, achieving a much more significant speedup of 12.7 times a non-parallelised test suite run on average. These findings indicate that the soundness and effectiveness of test suite parallelisation may be at odds with one another when order-dependent tests are present. This led them to suggest that developers should address the order-dependent tests in their test suites, using a tool such as ELECTRICTEST [15], allowing them to utilise the more powerful parallelisation strategies that were less accommodating of test order dependencies.

Lam et al. [91] proposed an algorithm for enhancing the soundness of test suite prioritisation, selection and parallelisation with respect to test order dependencies, again with the requirement that they are known and specified. As input, their algorithm takes the original test suite, a *modified* test suite, for example, a subset of the original test suite as produced by test selection, and a set of test order dependencies. The set of dependencies consists of a list of pairs of tests where the latter is impacted by the prior execution of the former. These are split into *positive dependencies*, where the first must be run before the second for the second to pass, and *negative dependencies*, where the second would fail if executed after the first. Using previous terminology, the former case represents a state-setter and a brittle and the latter case represents a polluter and a victim [141]. Examples of tools that could produce such a set include many of those examined in Section 2.5.2. As output, the algorithm produces an *enhanced* test suite — a test suite as close to the modified test suite as possible while respecting the specified test order dependencies. Initially, the algorithm

computes the set of tests that the tests of the modified test suite transitively depend upon from the specified positive dependencies. Iterating through the modified test suite, each test is inserted into to an initially empty enhanced test suite. This is done with the pre- and post-condition that the enhanced suite has all of its dependencies satisfied and the additional post-condition that the test is added to the end of the enhanced suite. To that end, the algorithm uses the previously computed set of dependencies to determine the sequence of tests that are needed to be executed before the test to be inserted into the enhanced suite. This sequence is sorted such that the order of the emergent enhanced suite will be as close to the optimised order of the input modified suite as possible. The dependent tests of this sequence are then iteratively inserted into the enhanced suite in a recursive manner, such that they also have their dependencies resolved (if any). To evaluate their algorithm, they performed prioritisation, selection and parallelisation upon a range of open-source Java subjects from previous work [90], using DTDETECTOR [175] to identify the dependency pairs. They measured the reduction in failures of order-dependent tests after enhancing both the modified developer-written and automatically generated test suites of their subjects. In the case of prioritisation, they observed an 100% reduction in failures for the developer-written test suites (i.e., all tests passed) and a 57% reduction for the automatically generated test suites. For selection, the result was 79% and 100%, respectively and for parallelisation, it was 100% and 66%.

User Interface Testing

Gao et al. [48] set out to investigate which external factors are the most important to control in an attempt to reduce flakiness in system user interface tests, performing an experiment upon five GUI-based Java projects. Specifically, they measured the impact of varying the operating system, the initial starting configuration, the time delay used by the testing framework (used to wait for a user interface to be fully rendered and stable before evaluating oracles) and the Java vendor and version, upon non-deterministic behaviour during test suite runs with respect to three application layers. For each layer, they derived metrics to measure the magnitude of the non-determinism across repeated test runs under various configurations of the investigated external factors. The lowest level layer was the *code layer* and refers to the source code of the software under test, where they measured the line coverage. To measure the magnitude of the variation in this layer, they calculated an entropy-based metric to measure the extent to which the line coverage was inconsistent over multiple test suite executions. The next layer was the *behavioural layer* which pertains to invariants regarding functional data, such as runtime values of variables and function return values, which are mined over a run of a test suite. Once again, they derived an entropy-based metric to measure how inconsistently invariants could be observed when repeating tests. The highest level layer was the *user interaction layer*, representing the interface state visible to the end user. They used a metric based upon the number of false positives observed from a GUI state oracle generator as a measure of the stability in this layer. From the findings of their experiments, they arrived at three overall guidelines for researchers and developers for promoting the reproducibility of test suite executions. The first was to ensure that the exact configuration of the application and the state of the execution platform when running a test suite is explicitly reported and shared. Referring to how some examples of non-determinism appeared unaffected by controlling the various external factors examined, the second was to run tests multiple times to accommodate these cases. The third was to use application domain information in an attempt to control test suite variability, giving an example of controlling the system time as an environmental factor when testing a time-based application.

In a related study, Gao et al. [49] presented an approach for ranking flaky test failures in the context of GUI regression testing based on their stability. The authors described the process of GUI regression testing as executing a regression test suite upon two versions of a GUI application, which invokes particular events to get the GUI into some particular state, where various properties (e.g., the coordinates and dimensions of elements such as text boxes) are measured and compared to identify discrepancies indicative of bugs. Given the flakiness of such tests, developers may face difficulty when identifying which discrepancies represent genuine faults as opposed to being artifacts of external factors such as window placement or resolution. Under the reasoning that

the most unstable properties are the least useful for finding bugs, they described an approach for ranking test failures based upon the entropy of the properties they compare. To calculate the entropy, the test suite must be repeatedly executed to arrive at a decent sample of values for each measured property. They evaluated their approach on three Java GUI applications with known bugs and found that their ranking technique indeed percolated the genuine, known bugs to within the top 30 of all the lists of discrepancies.

Mutation Testing

Shi et al. [139] investigated ways to mitigate tests with inconsistent coverage when performing mutation testing. In particular, they used 30 open-source Java projects to evaluate the effectiveness of various modifications to the Java mutation testing tool called PIT [262]. Specifically, they investigated the impact of repeating and isolating tests during the coverage collection and mutant killing runs of PIT. The purpose of the coverage collection run is to identify the set of program statements each test covers so that PIT can generate mutants which they have a chance of killing. Given the finding that 22% of statements were inconsistently covered in their motivating study, they reasoned that repeating tests improves the reliability of this process since a single run may not accurately represent which lines a test may cover. As for the motivation behind isolating tests at this stage, they referred to the potential for test order dependencies to impact coverage. They found an insignificant increase in the number of generated mutants when repeating and isolating tests for the considerable increase in the amount of time taken to do so, thus concluding that it may not be worthwhile. They went on to investigate the impact of repeating and isolating tests during the mutant killing run, citing the same motivations as before. In this run, tests are executed to identify which mutants they are able to identify by failing and thus kill. The authors found that repeating and isolating tests at this stage reduced the number of mutants with an unknown killed status (i.e., mutants that a test ought to cover based on the coverage collection run, but at the mutant killing stage, does not) by 79%, demonstrating the benefits of this approach.

Automatic Test Generation

Arcuri et al. [10] extended the search-based test suite generation tool called EVOSUITE [44] to improve the coverage and “stability”, a term they used to mean the absence of flakiness, of test suites generated for Java classes with environmental dependencies. Their multifaceted approach involved instrumenting the bytecode of generated tests to reset the state of static fields following executions and extending EVOSUITE’s genetic algorithm to generate test methods capable of writing to the standard input stream to accommodate classes which read user input. They also implemented a virtual file system, by mocking Java classes that perform input and output, that is reset after the execution of each test case. Finally, they mocked methods and classes which provide sources of environmental input or other non-determinism, such as the `Calendar` [208] and `Random` [267] classes, both of which make use of system time. An experiment using 30 classes of the SF100 corpus [44] known to lead to flaky generated tests indicated that the combination of these mitigations reduced the number of flaky generated tests from an average of 2.43 per class to 0.02.

Conclusion for RQ4.1: What methods and insights are available for mitigating against flaky tests? An early study proposed and evaluated a lightweight alternative to mitigating test order dependencies via process isolation, based on dynamically re-initializing modified values following test runs, finding it to introduce an average time overhead of only 34% compared to 618% [14]. In the context of test suite parallelisation, one study proposed and evaluated schedulers for making the most of available CPUs when executing test suites with known and specified order-dependent tests, reporting an average speedup of up to 7 times compared to a regular test suite run without breaking any test order dependencies [15]. Another source examined how best to configure concurrent execution when faced with order-dependent

tests, with the main finding being that ensuring consistent test outcomes appears to be a trade off with achieving the fastest test runs [22]. One experiment demonstrated the applicability of an algorithm for satisfying broken dependencies following rescheduling by prioritisation, selection or parallelisation, reducing the number of failed order-dependent tests in developer-written test suites by between 79% and 100% [91]. In the area of user interface testing, one study suggested three guidelines after studying the impact of various factors pertaining to the test execution environment and platform on the flakiness of tests. They recommended that the exact application configuration and state of the execution platform be explicitly reported, that it may be useful to re-execute tests several times to account for unknown or uncontrollable non-determinism, and that information regarding the application domain ought to be taken into account when attempting to control flakiness [48]. A study into mitigating flaky test coverage in mutation testing found that repeating and isolating test runs during the mutant killing phase reduced the number of non-deterministically killed mutants by 79% [139]. Finally, extensions to the automatic test generation tool called EVOSUITE [44] were found to significantly reduce the number of flaky tests it generated, from an average of 2.43 per test class to 0.02 [10].

2.6.2 Repair

With regards to repairing flaky tests, numerous sources have presented insights and strategies to assist developers in removing the flakiness from their test suites, as opposed to simply mitigating it. Some have mined insights from examining previously repaired flaky tests [37, 105, 136]. With the aim of helping developers improve the reliability of their test suites in the shortest amount of time, one study presented a technique for prioritizing flaky tests for repair [160]. Another deployed a survey with the aim of identifying the most important pieces of information, as rated by software developers, needed to repair flaky tests [37]. A line of research concerned with automatically identifying the root causes of flaky tests, in order to assist developers in repairing them, has emerged in recent years [88, 151, 178]. Finally, I examine techniques for the automatic repair of order- and implementation-dependent flaky tests and those that are flaky due to external data [42, 141, 174].

Insights from Previous Repairs

Studies have provided insights into repairing various kinds of flaky tests by considering previously applied fixes in general software projects. Luo et al. [105] examined commits that repaired flaky tests within projects from the Apache Software Foundation. Eck et al. [37] surveyed Mozilla developers about flaky tests they had previously fixed. The prevalence of the different types of repairs as reported by these two studies are summarised in Table 2.12. In terms of where these were applied, figures range from between 71% to 88% exclusively test code [37, 89, 105]. The remainder were applied to either the code under test, both the test code and the code under test or elsewhere, such as within configuration files. This finding indicates that the origin of test flakiness may not exclusively be test code, and in the cases where fixes were applied to non-test code, suggests that a flaky test has, possibly indirectly, led a developer to discover and repair a bug in the code under test. This provides an additional argument against ignoring flaky tests, as previously discussed in Section 2.4.1. With regards to repairing flaky tests of the *asynchronous wait* category, the most common fix involved introducing a call to Java's `waitFor` [263], or Python's `await` [210], or modifying an existing such call, accounting for between 57% and 86% of previous cases. The `waitFor` method blocks the calling thread until a specific condition is satisfied, or until a time limit is reached. This would be used to explicitly wait until an asynchronous call has fully completed before evaluating assertions, thus eliminating any timing-based flakiness. Similarly, Luo et al. identified fixes regarding the addition or modification of a fixed time delay, such as via a call to Java's `sleep` [277], to make up 27% of historical repairs. Compared to `waitFor`, `sleep` is a less reliable solution since the specified time delay can only ever be an estimate of the upper limit of the time taken for the asynchronous call to complete in any given test run. To that end, the authors identified that 60% of such repairs increased a time delay, suggesting that the developers

believed their estimate to be too low or perhaps too unreliable across different machines. A study by Malm et al. [107] found `sleep`-type delays to generally be more common than `waitFor`, via automatic analysis of seven projects written in the C language. Another solution was to simply reorder the sequence of events such that some useful computation is performed after making an asynchronous call instead of quiescently waiting for it to complete, accounting for between 3% and 13% of cases. Such a change does not actually address the root problem, but may be preferable to simply blocking the calling thread.

For fixing flaky tests of the *concurrency* category, the most common repair, according to Eck et al., again pertained to the introduction or modification of a call to `waitFor` or `await`, with a prevalence of 46%. Adding locks to ensure mutual exclusion between threads accounted for between 21% and 31% of fixes for flaky tests in this category. Between 9% and 26% of historical repairs involved modifying concurrency guard conditions, such as the conditions for controlling which threads may enter which regions of code at any one time. Making code more deterministic by eliminating concurrency and enforcing sequential execution made up between 5% and 25% of fixes. An additional type of repair as identified by Luo et al. consisted of modifying test assertions, and nothing else, to account for all possible valid behaviours in the face of the non-determinism permitted by the concurrent program, describing 9% of previous repairs.

Ensuring proper setup and teardown procedures between test runs was the most common fix for order-dependent tests according to Luo et al., accounting for 74% of cases. Another fix was to explicitly remove the test order dependency by, for instance, making a copy of the associated shared resource (e.g., an object or directory), making up between 16% and 100% of repairs. Another repair was to merge tests into a single, independent test by, for example, simply copying the code of one test into another, which described 10% of fixes for order-dependent tests according to Luo et al..

As part of an empirical evaluation into flakiness specific to user interface tests, Romano et al. [136] examined 235 developer-repaired flaky user interface tests across a sample of web and Android applications. They categorised the type of repair applied into five categories. The first category, *delay*, accounted for 27% of cases and was subcategorised into repairs involving the addition or increase of a fixed time delay or those relating to an explicit waiting call, as described previously. The second category, *dependency*, related to repairs regarding an external dependency, such as fixing an incorrectly called API function or changing the version of a library. The authors placed 8% of flaky tests into this category. The final three categories were *refactor test*, *disable features* (specifically disabling animations in the user interface, which was found to be a significant cause of flakiness), and *remove test*. These represented 32%, 2% and 31% of repairs respectively, the latter category not being a true repair, but simply eliminating the flaky test from the test suite.

Prioritizing Flaky Tests for Repair

Vysali et al. [160] presented GREEDYFLAKE, a technique for prioritizing the fixing of flaky tests based on the number of *flakily covered* program statements that would become *robustly covered* if the flaky test was repaired. They defined a flakily covered statement as one that is only covered by flaky tests and a robustly covered statement as one that is covered by one or more non-flaky tests. Their motivation for this approach was based on the notion that program elements covered exclusively by flaky tests are unlikely to be as well-tested as those covered by tests with more deterministic behaviour. They evaluated this technique with the test suites of three large open-source Python projects by comparing how quickly flakily covered program statements would become robustly covered when prioritizing flaky test repairs based on other schemes. Specifically, they compared their approach to random prioritisation (i.e., merely shuffling the list of flaky tests to be repaired), prioritisation based on total statement coverage, prioritisation based on newly covered statements (in a regression testing context) and prioritisation based on total number of flakily covered statements. Their evaluation demonstrated that GREEDYFLAKE outperformed all of the other methods for every subject with regards to identifying the best fixing order for reducing the greatest number of flakily covered program statements in the fewest number of flaky test fixes.

Table 2.12: Prevalence of different types of repairs for three prominent categories of flaky tests according to Luo et al. [105] and Eck et al. [37]. Dashes indicate that the study did not consider that type of repair.

Category	Repair	Luo et al. [105]	Eck et al. [37]
Asynchronous Wait	Add/modify <code>waitFor/await</code>	57%	86%
	Add/modify <code>sleep</code>	27%	-
	Reorder code	3%	13%
	Other	13%	1%
Concurrency	Add/modify <code>waitFor/await</code>	-	46%
	Add lock	31%	21%
	Modify concurrency guard	9%	26%
	Make code deterministic	25%	5%
	Modify assertions	9%	-
	Other	26%	2%
Test order dependency	Setup/teardown state	74%	-
	Remove dependency	16%	100%
	Merge tests	10%	-
	Other	0%	0%

Important Information for Repair

From a multi-vocal literature review, Eck et al. [37] identified eight pieces of useful information for fixing flaky tests. They asked developers via an online survey to rate on a Likert scale, for each piece of information, how important they considered it and how difficult they thought it was to obtain. The results of this survey are presented in Table 2.13. The most important of these, as considered by developers, was the context leading to the failure, which the developers also identified as the second hardest to obtain. As one developer described: “the most difficult operation is reproducing a flaky test, as sometimes only 1/20 fails”. Another respondent explained how the verification of the failing behaviour can be even more difficult due to the slowness of tests and the execution environment, among other factors, citing their slow UI tests as an example. The second most important was the nature of the flakiness, or its category, which the developers considered the most difficult to obtain. On this topic, one developer referred to identifying the root cause of a flaky test as “a big challenge”, due to the wide array of possible contributing factors such as concurrency issues or cache related problems. The remaining pieces of information in descending order of perceived importance were: the origin of the flakiness (i.e., whether it is rooted in test code of the code under test), the involved code elements, the changes needed to perform the fix, the context leading to the flaky test passing, the commit introducing the flakiness and the history of the test’s flakiness (i.e., previous causes and fixes).

Identifying Root Causes to Assist Repair

Having identified the nature and origin of flaky tests to be the most important pieces of information needed to repair flaky tests [37], several approaches have been developed to assist developers by revealing information about their root causes. One such approach, ROOTFINDER, was presented by Lam et al. [88] and compares attributes of the execution of flaky tests in their passing and failing cases. Their technique leverages the `Torch` framework for instrumenting API method calls. Specifically, the framework replaces method calls with a generated version, which calls the original method, but also provides three callbacks, one just before the call, one just after and one upon a raised exception. As part of ROOTFINDER, they implemented callbacks to measure properties including, but not limited to, the identifiers of the calling process, thread and parent process, the caller API, the timestamp of the call, the return value and any exception that was raised. During repeated test suite runs, the information recorded by these callbacks is stored in separate logs for passing and failing test executions. Using these logs, ROOTFINDER evaluates various predicates,

Table 2.13: Useful information for repairing flaky tests as rated on a Likert scale by developers with regards to importance and difficulty in obtaining, as found by Eck et al. [37].

Information	Importance	Difficulty
Context leading to failure	2.69	1.65
Nature of the flakiness	2.40	1.71
Origin of the flakiness	2.22	1.17
Involved code elements	2.21	1.13
Changes to perform the fix	2.08	1.59
Context leading to passing	1.95	1.20
Commit introducing the flakiness	1.89	0.75
History of the test’s flakiness	1.79	0.83

the outcomes of which are kept in *predicate logs*, again with one for passing tests and one for failing. Along with the values of these predicates, the code location of the instrumented method call, the calling thread id and an index that is incremented each time the method is called at the same location is also recorded to provide additional context, referred to as *epochs*. Examples of predicates that ROOTFINDER evaluates include whether the return value at some epoch is the same as all chronologically previous epochs (which is useful for identifying non-determinism), whether a specified amount of time is observed between a particular sequence of calls (which is useful for identifying thread interleavings), and whether the method call took longer than a specified upper limit. ROOTFINDER uses these predicate logs to perform further analysis to identify those that are indicative of flaky tests. Those that are always true in passing cases and always false in failing cases (or vice versa) indicate that a method consistently behaves differently in passing and failing runs, which they posited, may help to explain why a test is flaky by showing how the passing and failing executions differ. They went on to provide examples of methods to instrument and predicates to evaluate that may be useful in identifying examples of several of the categories of flaky tests identified by Luo et al. [105]. For instance, for the *asynchronous wait* category, the predicate should indicate that the asynchronous call always took longer in the failing runs such that some timing assumption did not hold (e.g., a fixed time delay was insufficient to wait for an asynchronous method and thus results in the failing test cases).

Ziftci et al. [178] presented another way to compare passing and failing executions. For a given flaky test, their approach performs some number of instrumented runs, collecting execution traces. For a given failing run of the flaky test, their technique identifies the passing execution with the longest common prefix and extracts the point at which the execution diverges into the failing case. They posited that by inspecting the code location of the divergence between passing and failing cases, developers might understand why their test is flaky. They implemented their approach as a tool in Google’s continuous integration platform, making use of its internal flakiness scoring mechanism to select tests that are flaky, but whose failing cases are not too rare, citing limited resources for repeated executions. To assess the applicability of their tool, they performed several case studies. One such study involved two developers fixing 83 flaky tests, that had previously been repaired by other developers, with the assistance of the tool. The developers reported that, in between 36% and 43% of cases, the divergent lines reported by the tool contained the exact code location that required repair. For between 25% and 32% of cases, the fix was applied to a different region of code, but the report was still helpful and relevant. In 15% of cases, the report was inconclusive, hard to understand or not useful.

Terragni et al. [151] proposed a new technique for identifying the root causes of flaky tests, motivated by two main limitations of ROOTFINDER [88]. First, the **Torch**-based instrumentation incurs some amount of runtime overhead which may be problematic for several reasons, such as potentially impacting the manifestation of time-sensitive flaky tests and increasing the overall analysis times. Second, a general limitation of any strategy based upon rerunning tests, is that it *passively* explores the non-deterministic space of test executions. In other words, repeatedly executing a test under the exact same conditions means that the more elusive flaky tests that

Table 2.14: Overview of three studies that proposed techniques for repairing specific types of flaky tests.

Study	Targets	Underlying Technique
Shi et al. [141]	Order-dependent tests	Delta debugging [172]
Fazzini et al. [42]	External dependencies	Component-based program synthesis [81]
Zhang et al. [174]	Implementation-dependent tests	Template-based repair [101]

are manifested only in rarely occurring scenarios, such as a particular execution order of multiple threads, are unlikely to be identified without a significant number of repeats. This led the authors to propose their technique, which they described as *actively* exploring this space. Specifically, their approach executes some test under analysis repeatedly in separate containers (i.e., via DOCKER [214]), each designed to manifest a particular category of flakiness. For example, one container attempts to manifest flaky tests of the *concurrency* category by varying the number of available CPU cores and randomly spawning threads that execute dummy operations to occupy the available resources. Another container attempts to identify order-dependent tests by arbitrarily executing other tests before the test under analysis, randomly taken from its containing test suite. As well as these, the test is executed in a baseline container, which makes no explicit attempt to manifest any particular kind of flakiness. After some upper limit of executions, the failure rate (i.e., the ratio of test failures to total runs) is calculated for each container and the category associated with the container that has the largest difference in failure rate compared to the baseline container is presented as the most likely cause.

In the domain of web applications, Morán et al. [111] proposed a technique named FLAKYLOC for identifying the root causes of flaky tests. Similar to the work of Presler-Marshall et al. [132], their technique consisted of repeatedly executing a Selenium [275] test suite under multiple configurations of a range of factors believed to impact the likelihood of manifesting test flakiness. These factors were: the operating system, the screen resolution, the web driver (i.e., web browser), the number of CPU cores, the network bandwidth, and the amount of available memory. In the case that a test fails under some configurations and not others, FLAKYLOC identifies the likely cause by calculating suspiciousness rankings for each factor, as would be done for source code lines in the context of fault localisation [21, 25, 157, 166].

Formalizing a test case as a series of discrete “steps”, Groce et al. [55] presented a modification to the delta debugging approach [172], a type of binary search, for minimizing the set of such steps required to exhibit non-determinism. Conceivably, this could help a developer to identify the specific cause of a flaky test. Their approach operates with two types of non-determinism. The first, that they term *horizontal non-determinism*, refers to a divergence in execution traces between two independent runs of a test case. The second, *vertical non-determinism*, describes the case where a supposedly idempotent operation, such as a file system operation that fails due to access permissions, returns inconsistent results following repeated executions in the same trace. Implementing their technique in Python, an evaluation with four subjects found that it reduced the number of steps in non-deterministic test cases by between 85% to 99%.

Automatic Approaches to Generate or Improve Repairs

Shi et al. [141] developed a tool for automatically repairing order-dependent tests, named IFIXFLAKIES, using the statements of other tests in the test suite, which they refer to as *helpers*. They identified two categories of order-dependent test, *victims*, which pass when run in isolation but may fail when executed after other tests, and *brittles*, which fail in isolation and thus depend on previous tests to be executed beforehand to pass. They refer to a subsequence of tests which appear to induce a failure in a victim as a *polluter* and those which result in a brittle passing as a *state-setter*. Furthermore, they refer to a sequence of tests executed between a polluter and a victim that appears to enable the victim to pass as a *cleaner*, in other words, it appears to “clean” the state pollution. Together with state-setters, these constitute the two types of helpers which they reasoned may contain the functionality needed to repair the order dependence in the

corresponding victim or brittle. Given an order-dependent test and an example of a passing and a failing test run order, their tool employs delta debugging [172] to identify a minimal sequence of tests that act as a polluter in the case of a victim, or a state-setter in the case of a brittle. In the case of a victim, IFIXFLAKIES will apply a delta debugging approach again to find a minimal cleaner for the identified polluter. Applying delta debugging once more, it identifies the minimal set of statements from the identified helper tests which, when arranged as a generated method called at the beginning of the order-dependent test (much like a set up method), induces it to pass in the previously failing order. To evaluate their framework, they recycled subjects from previous work [90], arriving at a subject of 13 Java modules. Their respective test suites contained a total of 6,744 tests, 110 of which were identified as order-dependent. With these order-dependent tests, IFIXFLAKIES was able to generate an average of 3.2 unique patches, with a mean size of 1.5 statements, that removed their order dependency. On average, it took only 186 seconds to generate a patch. They reported no cases where IFIXFLAKIES was unable to generate a patch for an order-dependent test. To further demonstrate the validity of their generated patches, they submitted them as pull requests to their respective projects. At the time of writing, 21 had been accepted by their developers.

Having categorised flaky tests discovered by Microsoft’s distributed build system, Lam et al. [89] identified the majority (78%) as being of the *asynchronous wait* category. This motivated them to specifically examine how such flaky tests had been previously repaired. By examining pull requests, they determined that the most common fix was related to changing the time delay between making the asynchronous call and evaluating assertions, accounting for 31% of cases. This further motivated them to propose the *Flakiness and Time Balancer* or *FaTB* technique for automatically improving such fixes by reducing their execution time. Their approach is essentially a binary search for the shortest waiting time that does not result in flakiness. Starting with the waiting time before the flaky test was fixed and the new waiting time that fixed the flaky test, the *FaTB* technique repeatedly bisects this interval and re-executes the test 100 times. For example, if a flaky test previously waited for 500ms before being adjusted to wait for 1000ms, *FaTB* would try 750ms. If this reintroduced flakiness after 100 repeats, then it would attempt 875ms. Otherwise if no flakiness was observed, it would try the middle point between no waiting and the previously successful time of 750ms (i.e., 375ms). This process then continues up to some upper limit of iterations. To evaluate this technique, they randomly sampled five applicable flaky tests from their dataset. In each case, their technique was able to find a waiting time that did not manifest flakiness and was shorter than the initial developer’s fix. However, for four of their sampled tests they never observed any flakiness at all, indicating that a larger set of tests may be needed to properly evaluate the technique.

Fazzini et al. [42] proposed a technique for the automatic generation of test mocks [145] in mobile applications. Test mocks replace real interactions between a mobile application and its environment (e.g. camera, microphone, GPS), a potential source of flakiness [152]. As such, their framework may not only improve the maintainability and performance of a test suite, but may also repair flaky tests. Their proposed technique, named MOKA, attempts to generate test mocks using a multi-stage strategy. Initially, MOKA leverages component-based program synthesis [81] to attempt to generate a test mock using a database of other mobile applications and their corresponding test suites. Should this fail, MOKA falls back to a “record and play” approach, which is to create a mock based on previous data recorded from the environmental interaction. On the same theme, Zhu et al. [177] proposed a machine learning based technique, named MOCKSNIFFER, for suggesting to developers whether a particular external dependency ought to be mocked or not within a test case. Following an empirical study involving four large Java projects, the authors devised ten rules to match cases where developers had decided to use mock objects when testing. These rules stemmed from several observations, for example, the tendency of developers to mock classes related to networked services or concurrency. Encoding a mocking decision as a tuple consisting of the test case, the class under test, the dependency (a class used in the test case but not in the class under test) and a label indicating whether the dependency was mocked or not, the authors devised 16 features to train their machine learning model, based on their earlier observations. Using the same four projects as their training set,

the authors built static analysers for each feature and evaluated several models, finding Gradient Boosting [45] to be the best. To evaluate their approach in the absence of similar techniques, they compared MOCKSNIFFER to three baselines: one based on existing heuristics from previous studies, another using the mock list used by EVOSUITE [40, 44], and a third based on the authors’ observations from their empirical study. Using an evaluation set of six projects distinct from their four training projects, the authors found MOCKSNIFFER to generally outperform the three baselines.

Extending on the work of NONDEX (see Section 2.5.1), Zhang et al. [174] proposed and evaluated a technique for repairing flaky tests arising from assumptions about non-deterministic or “underdetermined” specifications. Their approach, implemented as a tool named DEXFIX, uses an enhanced version of NONDEX to identify tests in need of repair, their failing assertion and the root cause of the flakiness. Their technique then applies one of four template-based repair strategies based on the nature of the non-determinism. The first is to replace the AssertJ assertion method `containsExactly` [203], for comparing collections, to an alternative method that only checks contents and not the order. This addresses flaky tests that expect a particular order for unordered collection type objects, such as sets or hash maps, in an assertion statement. The second is to replace the statement `new HashMap` [229] or `new HashSet` [230] with `new LinkedHashMap` [251] or `new LinkedHashMap` [252] respectively. These latter classes subclass their respective former, unordered classes but guarantee a deterministic iteration order. This strategy addresses flaky tests that expect a particular iteration order for objects of these types. The third strategy is to sort the output of the `getDeclaredFields` method [209], which is specified to return an array of the fields of a class but in no particular order. This addresses flaky tests that expect some kind of order, previously identified as a common cause of implementation dependent flakiness in Java projects [61, 140]. The final strategy addresses assertions that compare JSON strings [246] by replacing JUnit’s `Assert.assertEquals` [202] method, which performs an exact string comparison, with `JSONAssert.assertEquals` from the JSONassert project [247]. The latter method does not consider the order of JSON *objects*, a set of key/value pairs, and so is far less strict and brittle than a simple string comparison. Using NONDEX, Zhang et al. identified 275 flaky tests across 37 open-source Java projects from GitHub. Using DEXFIX, they were able to automatically repair 119. Of these generated patches, they submitted 102 as pull requests to the respective projects’ repositories. At the time of writing, 74 had been accepted, 23 were still pending and 5 had been rejected. One reason for rejection, the authors noted, was the developer’s hesitance to introduce a dependency on JSONassert.

Conclusion for RQ4.2: What methods and insights are available for repairing flaky tests?

By examining historical commits in Apache Software Foundation projects, and surveying Mozilla developers, two studies agreed that the most common type of fix for flaky tests of the *asynchronous wait* category involved a `waitFor` method or its equivalent [37, 105]. For those of the *concurrency* category, common repairs also involved `waitFor`-like constructs, as well as ensuring mutual exclusion via adding locks to concurrent code [37, 105]. For order-dependent tests, the most common fixes involved a test’s setup and teardown methods or explicit attempts to eliminate the dependency, by for instance creating a duplicate instance of some shared resource [37, 105]. A survey asking software developers to rate the most important pieces of information needed for fixing a flaky test, and the difficulty in obtaining them, identified the context leading to failure as the most important and the nature of the flaky test as the most difficult to obtain [37]. To that end, one study presented the ROOTFINDER tool that compares attributes from the execution of tests in passing and failing cases to identify any differences that could indicate the root cause of flakiness [88]. Another technique employs multiple containers that each attempt to manifest some specific category of flakiness, with the category associated with the container with the greatest difference in failure rate, as compared to a baseline, considered the most likely cause [151]. For the automatic repair of order-dependent flaky tests, one source presented an approach using program statements from elsewhere in

Table 2.15: The top ten most cited studies.

Author	Paper	Cited by
Luo et al.	An Empirical Analysis of Flaky Tests [105]	251
Shamshiri et al.	Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges [138]	160
Martinez et al.	Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset [108]	160
Zhang et al.	Empirically Revisiting the Test Independence Assumption [175]	126
Memon et al.	Taming Google-Scale Continuous Testing [110]	105
Hilton et al.	Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility [71]	105
Bell et al.	DeFlaker: Automatically Detecting Flaky Tests [16]	77
Bell et al.	Unit Test Virtualization with VMVM [14]	76
Vahabzadeh et al.	An Empirical Study of Bugs in Test Code [155]	73
Gyori et al.	Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency [62]	66

Table 2.16: The top five most prolific authors.

Author	Papers
Darko Marinov [16, 61, 62, 70, 71, 90, 92, 93, 105, 139–142, 163, 174]	15
August Shi [35, 61, 62, 90, 91, 127, 139–142, 174]	11
Wing Lam [88–93, 141, 163, 175]	9
Jonathan Bell [8, 14–16, 47, 70, 93, 117, 139]	9
Michael Hilton [8, 16, 33, 70, 71]	5

the test suite, generating fixes for 110 order-dependent tests, with 21 accepted by developers through pull requests [141]. For mobile applications, another study presented a framework for automatically generating test mocks, which could potentially repair flaky tests of several categories, including network, concurrency, and randomness. Another study presented DEXFIX [174], a template-based repair technique for automatically repairing implementation-dependent flaky tests, such as those detected by NONDEX [61, 140]. Of 275 flaky tests across 37 open-source Java projects, DEXFIX could repair 119, the generated patches of 74 of which had been submitted as pull requests to their respective repositories and merged by their developers.

2.7 Analysis

This section takes a high-level view of the studies examined in this survey. I identify the most prominent papers and authors and examine the various individual threads of research in the area, offering insights to those readers who want to familiarise themselves with the field. Furthermore, I provide a roundup of all the well-known tools, as examined in this survey, for detecting, mitigating, and repairing flaky tests and consider the various sets of projects used as subjects in their evaluations. Finally, I suggest future directions for research on flaky tests.

2.7.1 Prominent Papers and Authors

Table 2.15 shows the top ten most cited papers examined in this survey. This data is provided by Google Scholar [228] and is accurate as of the 7th of May, 2021. The most cited paper is *An Empirical Analysis of Flaky Tests* published in 2014 by Luo et al. [105]. Furthermore, it is cited by the majority of my sample of papers published after 2014, with two describing it as “seminal” [128, 159]. For these reasons, I consider it essential reading for people who want an introduction to

the field of flaky tests. The fourth most cited paper *Empirically Revisiting the Test Independence Assumption*, was the first to compare different strategies for detecting order-dependent flaky tests, and to present an automated tool for doing so [175]. This prefaced a line of further studies and a range of related tools [15, 17, 62, 90]. Several of the top ten papers are not specifically about flaky tests, but discuss a negative impact of flaky tests, such as *Taming Google-Scale Continuous Testing* [110]. I would recommend anyone beginning research into flaky tests to read these papers for a solid foundation in the research area.

The top five most frequent authors in the set of examined studies are listed in Table 2.16. At the top is Darko Marinov [212]. Marinov has been active in the area of flaky tests since its early beginnings, being a coauthor of *An Empirical Analysis of Flaky Tests* [105]. Along with his students, August Shi [204] and Wing Lam [282], second and third in the list respectively, he has been involved in the development of several automatic tools for detecting flaky tests, including ID-FLAKIES [90], POLDET [62] and NONDEX [61, 140]. Along with other collaborators, they were the first to develop and evaluate a fully automatic approach for repairing order- and implementation-dependent flaky tests [141, 174]. At number four on the list, but joint third with Lam, is Jonathon Bell [245], who has also been involved in the creation of numerous flaky test detection techniques such as ELECTRICTEST [15], PRADET [47] and DEFLAKER [16]. Bell frequently coauthors with Michael Hilton [255], at number five on the list, who has been particularly active on test flakiness in the context of continuous integration [33, 71]. I would direct those who want to read more about flaky test research to the publications of these five authors.

2.7.2 Tools and Subject Sets

Table 2.17 provides a complete list of all the named tools for detecting, mitigating and repairing flaky tests examined in this survey. To evaluate the performance of these tools, a number of sets of projects have been created as subjects and in several cases have been reused in multiple studies. Zhang et al. [175] used four open-source Java projects, containing a combined total of 4,176 developer-written tests, to compare the performance of various configurations of DTDETECTOR, their order-dependent test detection tool. These four subjects were then reused as part two further evaluations of later tools, namely ELECTRICTEST [15] and PRADET [47], and, along with the results of these three evaluations, are presented in Table 2.10. Bell et al. [16] presented a set of 5,966 commits across 26 open-source Java projects comprising 28,068 test methods. They used these to evaluate the performance of DEFLAKER, their flaky test detection tool. The DEFLAKER subject set was also used by Pinto et al. to train various classifiers using natural language processing techniques in an attempt to statically identify flaky tests [128]. Projects from this subject set, along with various other sources, such as popular projects on GitHub, were combined by Lam et al. [90] to create two subject sets for their evaluation of their own detection tool called IDFLAKIES. The first of these two sets, called the *comprehensive* set, contained 183 open-source Java projects. The second of these, called the *extended* set, consisted of 500 further projects disjoint from the *comprehensive* set. The projects of the *comprehensive* set were then reused in later studies involving Lam [89, 91, 93, 141]. Flaky tests identified from these subject sets were later called the *Illinois Dataset of Flaky Tests* [8, 163].

Table 2.17: A list of named techniques examined in this study.

Technique	Description	Source	Section
DTDETECTOR	Detects order-dependent tests by either reversing the test suite order, shuffling it, executing every k -permutation in isolation or only executing permutations that are likely to expose a test order dependency, based upon conservative analysis of static field access between tests and monitoring file usage.	[175]	§2.5.2
ORCALEPOLISH	May indirectly identify order-dependent tests, or tests that have the potential to become order-dependent, by detecting brittle assertions using taint analysis.	[74]	§2.5.2

VMVM	Implements a lightweight alternative to full process isolation for mitigating possible test order dependencies by only reinitializing classes containing static fields that may facilitate state-based side effects between test runs.	[14]	§2.6.1
ELECTRICTEST	Refines the dependency analysis of DTDETECTOR by considering aliasing, using an approach that leverages garbage collection and object tagging. Unlike DTDETECTOR, ELECTRICTEST does not verify the order-dependent tests it identifies, i.e., by executing them, meaning that it may give many false positives.	[15]	§2.5.2
POLDET	Models heap memory as a multi-rooted graph, with objects, classes and primitive values as nodes, fields as edges and static fields of all loaded classes as roots. Computes and compares heap graph before the setup phase and after the teardown phase of each test run to identify any observable changes in program state or "pollution", to arrive a list of polluting tests, i.e., tests that may cause (but may not themselves be impacted by) test order dependencies.	[62]	§2.5.2
NONDEX	Manifests flaky tests caused by the assumption of a deterministic implementation of a non-deterministic specification by reimplementing a range of methods and classes in the Java standard library to randomise the non-deterministic elements of their respective specifications.	[61, 140]	§2.5.1
DEFLAKER	Given a new version of the software under test, i.e., after a commit, detects flaky tests as those whose outcome changes despite not covering any modified code.	[16]	§2.5.1
PRADET	Follows a similar methodology as ELECTRICTEST, but verifies its findings by modelling a test suite as a graph, with tests as nodes and possible dependencies as edges, and proceeds to invert each edge and schedule a corresponding test run to identify if the respective dependency is manifest, i.e., if it affects the outcome of the dependent test.	[47]	§2.5.2
ROOTFINDER	Collects and processes information recorded from passing and failing executions of repeated test runs via instrumented method calls. Aims to assist developers in understanding the causes of flaky tests by, for example, showing the differences in the execution environment between passing and failing cases.	[88]	§2.6.2
TEDD	Detects order-dependent tests in web applications by considering dependencies facilitated by persistent data stored on the server-side and implicit shared data via the Document Object Model on the client-side, rather than by internal Java objects.	[17]	§2.5.2
IDFLAKIES	Classifies flaky tests as being order-dependent or not based upon comparing the outcomes of repeated executions of failing tests in a modified test run order to the original test run order.	[90]	§2.5.2
IFIXFLAKIES	Generates fixes for an order-dependent test from the statements of other tests in the test suite that have been identified as "helpers", i.e., that enable the dependent test to pass when executed previously.	[141]	§2.6.2
FLASH	Targeting Machine Learning applications, mines test suites for approximate assertions and repeatedly executes the containing tests to arrive at a sample of actual values evaluated within them. From these samples, FLASH estimates the probability that an assertion will fail and declares a containing test flaky if it is above a threshold.	[35]	§2.5.1
FLAST	A purely static approach to flaky test detection, represents tests using a bag of words model over the tokens of their source code and trains a k-nearest neighbour classifier to label them as flaky or not.	[159]	§2.5.1

MOKA	Uses component-based program synthesis to automatically generate test mocks for mobile apps, thereby potentially repairing flaky tests caused by dependence upon external data.	[42]	§2.6.2
SHAKER	Introduces CPU and memory stress during repeated test suite executions in an attempt to increase the probability of manifesting flaky tests of the <i>asynchronous wait</i> and <i>concurrency</i> categories.	[144]	§2.5.1
DEXFIX	Uses template-based automatic program repair to fix implementation-dependent flaky tests, such as those detected by NONDEX.	[174]	§2.6.2
FLAKEFLAGGER	A technique for detecting flaky tests without requiring test suite reruns, using a machine learning model. Requires a combination of dynamic test data, such as line coverage, and static data, such as features of the test’s source code.	[8]	§2.5.1

2.7.3 Related Research Areas

In many instances, the flaky test literature is tangential with several other research areas. An index of these are given in Table 2.18, with relevant citations from the set of collected papers, and references to the sections in this survey that mention them. Emerging as the area with the strongest intersection is continuous integration research. The associated citations pertain to the evaluation of the prevalence and impacts of test flakiness with respect to continuous integration systems. Since such systems involve a vast number of test executions, it is perhaps unsurprising that there is a strong overlap between this area of study and flaky test research. Test acceleration research, producing techniques that often change the test execution schedule, such as test suite selection [97, 106, 142], prioritisation [67, 127, 169], reduction [9, 100, 158] and parallelisation [15, 22, 66], is another area with a significant relationship with flaky tests.

Continuous Integration

Flaky tests have been studied several times in the context of continuous integration systems. An early source described an account of how transitioning to a continuous integration system led to developers observing the true scale of the flakiness in their test suites, since tests were being executed more often [87]. Since then, continuous integration systems have been a frequent object of study due to the vast amount of test execution data they make available. Labuschagne et al. [86] studied the build history of open-source projects using Travis and found that 13% of a sample of transitioning tests were flaky. Hilton et al. [71] deployed a survey asking to developers to estimate the number of continuous integration builds that failed due to true test failures and due to flaky test failures, finding no significant difference between the two distributions. Memon et al. [110] analysed Google’s internal continuous integration system and found flakiness was to blame for 41% of “test targets” that had previously passed at least once and failed at least once. Microsoft’s distributed build system has also been an object of study several times, with one study finding that had it not automatically identified and filtered the 0.02% of sampled test executions that were flaky, they would have gone on to cause what would have been 5.7% of all failed builds, demonstrating the impact that a relatively small number of flaky tests can impose [89]. Finally, Durieux et al. [33] found that 47% of previously failing builds that had been manually restarted went on to pass, suggesting that they may have initially failed due to flaky tests.

Test Acceleration

A specific category of flaky tests, order-dependent tests, have been identified as a potential burden to the soundness of much research in the area of test acceleration. This is because they may produce inconsistent outcomes when their containing test suites are reordered or otherwise modified. Zhang et al. [175] examined how many test cases gave inconsistent outcomes when applying five different test prioritisation schemes. Bell et al. [15] proposed a scheduler for achieving sound parallelisation when test suites contain order-dependent tests, though this was found to be at the cost of a

Table 2.18: Connections between related areas and the field of flakiness, with citations and references to relevant sections in this paper.

Area	Reference	Section
Continuous integration	[33, 70, 71, 86, 87, 89, 97, 110, 142]	§2.3.1, §2.4.2, §2.6.2
Test acceleration	[15, 17, 22, 47, 91, 175]	§2.3.2, §2.4.2, §2.5.2, §2.6.1
User interface testing	[48, 49, 132, 136, 169]	§2.3.3, §2.4.2, §2.4.3, §2.6.1
Software engineering education	[132, 140, 146]	§2.4.3
Automatic test generation	[10, 125, 138]	§2.4.3, §2.6.1
Mutation testing	[68, 139]	§2.4.3, §2.6.1
Automatic program repair	[108, 167]	§2.4.3
Machine learning testing	[35, 116]	§2.3.3, §2.5.1
Fault localisation	[157]	§2.4.3

considerable degree of efficiency. Candido et al. [22] examined the degree to which order-dependent tests were manifested when applying several different test suite parallelisation techniques. Lam et al. [91] proposed an algorithm for ensuring order-dependent tests had their dependencies satisfied after applying test acceleration, while attempting to minimise the loss of speedup in the test suite execution process.

2.7.4 Limitations and Future Directions

In this literature review, I examined many empirical studies on flaky tests [33, 59, 105, 152, 155, 175]. However, there is relatively little focus on the views and experiences of software developers. Since flaky tests are primarily a developer problem, there is an underutilised opportunity to acquire valuable insights from those who experience them first-hand. This is important to ensure that future research is relevant developers. Hilton et al. [71] and Eck et al. [37] both conducted developer-oriented studies concerning flaky tests. However, both suffer from organisational and self-reporting bias [31]. This is because they both focus on a single organisation and both take the form of questionnaires and interviews. In Chapter 3, I address these limitations by conducting a developer study involving both a questionnaire and an analysis of StackOverflow threads about flaky tests. Because the respondents to the questionnaire are from many different organisations, the results are unlikely to be skewed by organisational bias. Analysing StackOverflow threads mitigates self-reporting bias because this does not involve developers recounting their own experiences.

In Section 2.5.1, I examined several studies that use machine learning classifiers to detect flaky tests [4, 8, 82, 128, 159]. However, there remains little research into the features used to encode the test cases. Furthermore, none of these studies consider the detection of order-dependent flaky tests. As discussed in Section 2.4.2, order-dependent flaky tests can cause unique and significant difficulties for both developers and researchers. Therefore, it is worthwhile to consider their detection via machine learning classifiers. In Chapter 3, I address this research gap by proposing a new feature set for encoding test cases in machine learning-based flaky test detection. An evaluation involving 26 projects shows that classifiers based on this feature set outperform the previous state-of-the-art [8] when detecting both order- and non-order-dependent flaky tests.

In Section 2.5, I described a range of automated techniques for detecting flaky tests. Many are based on rerunning test cases, such as IDFLAKIES [90]. The main drawback of these techniques is the excessive number of repeated test case executions. This makes them expensive and likely impractical for deployment in large software projects. Another approach is to use a machine learning classifier, such as FLAKEFLAGGER [8]. While this is significantly faster than any rerunning-based technique, the accuracy of the classifiers can be questionable. Addressing both of these limitations simultaneously, I introduce CANNIER in Chapter 5. An empirical evaluation involving 30 open-source projects demonstrates that CANNIER can achieve a good balance between high detection performance and low run time cost when detecting flaky tests.

On the topic of machine learning-based detection, another limitation of many studies is that

they evaluate techniques using hold-out sets of test cases from the same projects the classifiers were trained with. Furthermore, they do not consider how well the techniques apply to future test cases as projects evolve. Both of these factors are important to consider when evaluating the generalisability of any such technique. Addressing these shortcomings, in Chapter 6 I introduce FLAKEFRIEND. Building upon the successes of CANNIER, this automated technique classifies existing test cases in a given project and produces a reusable classifier for the future test cases in that project. Furthermore, the empirical evaluation of FLAKEFRIEND uses hold-out sets of test cases from projects that were not involved in training, offering a reliable assessment of the technique’s inter-project generalisability.

On a more general note, Harman and O’Hearn [68] proposed the aphorism “Assume all Tests Are Flaky” (ATAF), thereby taking the view that all tests have the potential to be non-deterministic and that testing methodologies ought to accommodate for this, rather than to assume that tests are generally deterministic and that flaky tests are an exception. Overall, the general thesis of ATAF is to move away from efforts to control and mitigate flakiness, in the hope of making testing fully deterministic, and to move towards a situation where flaky tests are considered an unavoidable aspect of testing that are fully considered and accommodated. To that end, the authors proposed five open research questions regarding the assessment, prediction, amelioration, reduction and general accommodation of test flakiness.

Assessment

The first research question considers the possibility of an approximate measurement of the flakiness of a test case that goes beyond simply classifying it as flaky or not. So far, studies have measured flakiness using entropy [49, 84] and recently one has defined its own *flakiness-ratio* (FR) calculated for a test case τ as $FR(\tau) = 2 * \min(\tau_P, \tau_F) \div (\tau_P + \tau_F)$, where τ_P and τ_F are the numbers of passing and failing runs of τ respectively [157]. Another study considered the failure rates of flaky tests when executed under different settings, drawing various conclusions about the probability of a flaky test manifesting itself and how many repeated test runs are likely to be necessary. Their finding that certain test run orders can lead to different failure rates for the same test, as opposed to just always failing or always passing, suggests that the binary classification of a flaky test as order-dependent or not may be insufficient, providing further motivation for a more continuous assessment of flakiness [92].

Prediction

Their second research question concerns the development of predictive models for flaky tests, such that they may be identified without having to execute them repeatedly. An attempt at fitting a Bayesian Network model to predict flakiness based on a diversity of test features, such as code complexity and historical failure rates, demonstrated mixed results [82]. Machalica et al. [106] trained a model to predict if tests would fail, given a set of code changes, that was used as part of a test selection system. It is conceivable that a similar approach might be useful for predicting if a test might fail flakily. An emerging thread of research has been concerned with the application of machine learning models and natural language processing techniques for the prediction of test flakiness based on purely static features of the body of the test case [128, 159]. Recently, Alshammari et al. [8] combined these static features with dynamic features of a test case, such as its line coverage, demonstrating that the combination of both feature sets considerably improves the model’s predictive effectiveness. However, their results demonstrated that there is still some way to go before the development a predictive model that would be reliable enough for practical use by developers.

Amelioration

Their third question asks if one can find ways to transform test goals so that they yield stronger signals to developers when tests are flaky. For example, in the context of regression testing, it

may be desirable to apply test acceleration techniques such as selection or prioritisation to reduce the amount of time taken to receive useful feedback from the runs of a test suite. In the past, objectives have included the selection/prioritisation of tests based on the coverage of modified program elements, runtime and energy consumption [168]. Under ATAF, a new objective could be the degree of test determinism, to achieve greater certainty sooner in the test run. To that end, a reliable measure of test flakiness is required, as per their previous research question regarding the assessment of flaky tests.

Reduction

Their fourth question regards how to reduce the flakiness of a test, or more abstractly, to reinterpret the signal of a flaky test in some way that makes flakiness less of an issue. The authors then go on to describe the possibility of an abstract interpretation approach [28] for testing and verification such that an increased degree of abstraction reduces the degree of variability and non-determinism in test outcomes. A possible realisation of this, to potentially reduce the flakiness stemming from the algorithmic non-determinism commonly seen in machine learning projects [35, 116], could reinterpret the outputs of probabilistic functions as parameterised probability distributions instead of concrete values. An assertion which then checks if an output is within an acceptable range, a type of oracle approximation [116] linked to flakiness when too restrictive [37], would instead pass if the probability of observing a value in that range within the abstractly interpreted distribution was above some acceptable threshold. Such a reinterpretation could eliminate the flakiness when corner cases are outside of the assumed range of acceptable outputs.

Reformulation

Their fifth and final question asks how the various techniques impacted by flaky tests, such as those examined in Section 2.4, could be reformulated to cater for non-deterministic tests. To that end, one recent study proposed reformulations of various fault localisation metrics that are “backward compatible” with their original forms but are able to accommodate flaky test outcomes [157]. Beyond flaky outcomes, the coverage of tests has also been observed to be inconsistent [48, 70, 139]. This potentially motivates the need for the reformulation or generalisation of techniques which depend on coverage information, such as fault localisation again, and also mutation testing [139] and search-based automatic test generation [79]. In the latter case, the coverage of automatically generated tests is often used as part of a fitness function that guides a search algorithm such as hill climbing, simulated annealing or an evolutionary search [109]. Given that the coverage of tests, automatically generated or otherwise, may not always be as deterministic as perhaps initially thought, more abstract measures of test suite quality may be preferable.

2.8 April 2023 Update

In April 2023, I repeated the paper collection methodology described in Section 2.2. I found a further 32 relevant studies, according to the inclusion and exclusion criteria in Table 2.1, published between April 2021 (when I performed the initial paper collection) and April 2023. In this section, I update my answers to the four research questions posed in Section 2.2 with the new findings presented in these papers. This section did not appear in the published article associated with this chapter, because it was published in October 2021 [120].

2.8.1 RQ1: What are the causes and associated factors of flaky tests?

Developer Studies

Ahmad et al. [5] performed an empirical analysis of developers’ perceptions of test flakiness. They set out to investigate the root causes of flakiness in closed-source industrial projects and how the developers address them. They also aimed to discover what factors developers perceive

as affecting flakiness and if these factors can determine if a test case is flaky. To do so, they conducted surveys, online workshops, site visits, and physical workshops at five companies. The survey results indicated asynchronous waiting, and configuration and dependency issues, were the most common causes. The authors concluded that many of the factors identified by developers as being associated with a non-flaky test were simply properties of good test cases, such as a small and simple test case with a single responsibility.

Habchi et al. [63] conducted a qualitative study involving interviews with 14 practitioners with a diverse range of roles, experience, and domains. One of their research questions concerned where flakiness typically originated from in a software system. Their results indicated that, in addition to test cases themselves, flakiness stems from poor orchestration between system components, the testing infrastructure, and external factors such as the operating system and firmware. This led the authors to recommend that future studies should consider these sources of flakiness rather than focusing solely on factors related to the test case code and the code under test.

Gruber et al. [56] also conducted a study involving developers, but their's involved a questionnaire rather than interviews. They received 335 responses from two populations of participants. The first were recruited from a particular automotive company, from which they received 102. The second were recruited globally using Prolific, and online service for recruiting subjects for scientific studies, from which they received 301. One of their research aims was to establish the most common causes of flaky tests. They identified concurrency and test order dependency as the most common, broadly following the results of previous work [105]. They also identified uninitialised variables, compiler differences, and unexpected API behaviour as uncommon causes.

Open-Source Projects

Hashemi et al. [69] conducted an empirical study of flaky tests in JavaScript projects. Following a similar methodology to Luo et al. [105], they categorised the causes of the flakiness in 481 commits that repaired flaky tests within the top-40 JavaScript projects on GitHub in terms of number of stars. The top-three categories were concurrency, asynchronous waiting, and operating system, accounting for 74, 70, and 66 commits respectively. In the context of this study, concurrency and asynchronous waiting had the same meaning as used in previous empirical studies [59, 105, 152]. The authors described the operating system category as test cases that fail on a specific operating system (or specific version of an operating) due to an implicit dependency on features or environmental factors specific to a particular to that operating system.

Costa et al. [27] analysed 741 open issues related to flakiness in GitHub projects written in C, Go, Java, JavaScript, and Python. Overall, they found that the top-three most common causes were asynchronous waiting, accounting for 103 issues, concurrency, representing 97, and platform dependency, responsible for 87. However, the results varied slightly by programming language. For C, the top-three were resource leak, platform dependency, and concurrency. Because C requires manual memory management, it is not surprising that resource leak was the most common. For the remaining languages, the most common causes were broadly similar, mostly asynchronous waiting, concurrency, platform dependency, and network with varying distributions.

Conclusion for RQ1: What are the causes and associated factors of flaky tests?

Developer studies indicate that the most common causes of flaky tests broadly align with previous empirical studies [37, 105], with root cause categories such as asynchronous waiting and test order dependency being common. They also identified causes that were previously unreported in the literature or only rarely mentioned, including configuration issues, poor orchestration between system components, and compiler differences [5, 56, 63]. An empirical study on test flakiness in C, Go, Java, JavaScript, and Python projects found that the most common causes of flaky tests are broadly similar between languages with the exception of C [27].

2.8.2 RQ2: What are the costs and consequences of flaky tests?

Developer Studies

As part of a qualitative study involving interviews with 14 developers, Habchi et al. [63] aimed to understand practitioners' perceptions of the impacts of flaky tests. The authors remarked how the impacts of flakiness are typically reported through the lens of industrial reports and empirical studies, rather than through direct discussions with practitioners. As well as wasting developers' time and disrupting continuous integration (CI), as has been extensively reported by other studies [33, 122], they found that flaky tests negatively impact testing practices, prompting developers to write fewer test cases to avoid problems in the worst-case scenario. They also found that flaky tests can undermine system reliability and disguise bugs by confusing developers, leading them to question the test suites ability to detect bugs and even ignore failures.

Gruber et al. [56] conducted a related study involving a questionnaire with 335 developer respondents. One of their research questions concerned the negative effects of flaky tests. Similarly to Habchi et al. [63] they found eroding trust in test suites, wasting developers' time, and hindering CI were the foremost costs of flakiness.

No-Fault-Found Test Failures

Rehman et al. [135] performed a case study involving a radio base station system at Ericsson. They were interested in test failures that did not lead a developer to identify a fault, also known as "No-Fault-Found" (NFF) test failures. The authors remarked that flaky tests are one of the most common causes, among others. Their dataset consisted of 9.9 million test runs of 10,383 test cases over four releases of the system. Each release is tested for 180 days before deployment, with the test runs in the second half (day 90 to day 180) defined as the "stable period" where new development is restricted. Their first research question aimed to quantify how often each test case failed without identifying a fault. They defined the "NFFRate" of a test case during a particular release as the ratio of the number of test runs that failed without identifying a fault to the total number of test runs. They defined the "StableNFFRate" of a test case during a release as the NFFRate of that test case during the stable period of the previous release. They found that between 24% to 36% of test cases never had a NFF failure, 56% to 76% to have had fewer than 5 NFF failures in 100 runs and 18% to 22% to have had more than 10. Their second research question concerned the number of test cases that had been executed a sufficient number of times for statistical confidence in their outcome. Using the standard sample size formula and the StableNFFRate of a test case, the authors estimated that between 78% and 83% of test cases had been executed enough times for 95% confidence in their results at the end of each release cycle. Their subsequent investigations concerned the number of test cases to exhibit instability during a release, the number of test failures that are likely to be NFF failures that can be deprioritised for investigation, and how to prioritise test reruns to find test instabilities sooner.

Simulating Flakiness

Cordy et al. [26] introduced FLAKIME, a tool for simulating flaky failures to assess their impact on automated software engineering techniques. The tool instruments the bytecode of test cases and can trigger flaky failures at any execution point in the test case, termed *flake points*. To simulate a flaky failure, FLAKIME raises an unchecked exception that would cause the test case to fail. The exception is guarded by a probabilistic condition that can be tuned to control the volume of flaky failures. Using Defects4J, the authors used their tool to assess the impact of flakiness on mutation testing and automated program repair. For mutation testing, they found that a small degree of flakiness can affect the mutation score, but as the degree of flakiness increase, the mutation score follows an asymptotic growth where increasing flakiness has a diminishing effect. This led them to conclude that flakiness is a potential issue for mutation testing but with limited impact. They found the impact of flakiness on automated program repair to be more significant, with a

considerable drop in the number of patches such techniques were able to produce after introducing artificial flakiness.

Conclusion for RQ2: What are the costs and consequences of flaky tests? Developer studies indicate that flaky tests have many negative consequences. They waste developers' time, erode developers' trust in test suites, disrupt CI, and negatively impact testing practices [56, 63]. An empirical study on test failures that do not lead developers to discover a fault, of which flaky tests are one of the most common causes, estimated that between 78% and 83% of the test cases of a proprietary project at Ericsson had been executed enough times for 95% confidence in their results [135]. Cordy et al. [26] introduced FLAKIME, a tool that simulates flaky failures, and demonstrated the impacts of flaky tests on mutation testing and automated program repair. They found the negative impacts to be considerably worse for automated program repair than for mutation testing.

2.8.3 RQ3: What insights and techniques can be applied to detect flaky tests?

Continuous Integration and Version Control

Lampel et al. [94] evaluated the suitability of machine learning classifiers for detecting “orange” builds at Mozilla. Orange builds are Mozilla’s term for CI builds that fail due to flaky tests. They explained that typically after a Mozilla developer pushes changes and triggers a CI build that fails, a “sheriff” will manually classify the failure as due to a genuine bug or due to flakiness (an orange build). Therefore, their approach of using classifiers automates the job of the sheriff. As subjects, they mined telemetry data from nearly two-million historical builds from 20 projects. This data contained the sheriffs’ verdicts about whether each build was flaky, providing the ground-truth labels. They encoded each build using a mixture of categorical and numeric features, including the total run time, average CPU load, and platform information. They split their dataset at a particular time point such that they had a test set consisting of 85% and an testing set of 15%. As the classifiers, they evaluated various tree-based ensemble models such as random forest [18]. They used their training set for both classifier training and hyper-parameter optimisation. When evaluated on their testing set, they observed a mean F1 score of approximately 0.75.

Ahmad et al. [2] evaluated a machine learning-based technique to determine if a specific *test failure* was flaky or not. Their approach, named MDFLAKER, requires a project to have version control (e.g., Git) and CI (e.g., TravisCI) setup. MDFLAKER encodes test cases for classification by a K-Nearest Neighbors (KNN) model using features based on trace-back coverage, flaky frequency, test smells, and test case size. The core idea behind trace-back coverage is to locate the changes in the code under test and identify if the failed test case covered those changes. The authors defined flaky frequency as the ratio of the number of previous transitions between passing and failing in the test case’s history to the total number of failures. The test smells the authors were interested in were associated with several well-established causes of flakiness, such as fixed-time sleeps in the test case code. The authors measured test case size in terms of the number of non-commented physical lines of test case code. The authors evaluated MDFLAKER using three open-source Python projects. They selected 212 versions of these projects that had at least one failing test, leading to a set of 2,166 test failures. To establish a ground truth, they executed each one 30 times to determine which were flaky failures and identified 1,372 as flaky. They split their dataset into 75% for training and 25% for testing and repeated the evaluation of the KNN classifier for every combination of the four categories of features. With all features, the KNN achieved an F1 score of 0.87.

Gruber et al. [58] performed an empirical study into detecting flaky tests using static features derived from version control and CI histories. As their evaluation subject, they used a large automotive software project written primarily in C and C++ and consisting of nine repositories organised as Git submodules. The project is configured with a CI system that automatically reruns test cases on weekends, when very little development takes place. This enables the system

to identify and label flaky tests. From the CI history, the authors randomly sampled the data for 100 flaky and 100 non-flaky “test units”. In this context, a test unit refers to a specific test case at a specific time. For each test unit, they collected the test outcomes and durations from the last three months. This information, combined with version control data from Git, enabled the authors to compute features about each test unit. They computed the flip rate of each test unit, a ratio based on the number of times the outcome has changed over a specific window of runs. The flip rate can be configured with a decay function to apply smaller weights to older runs and larger weights to newer runs. They also calculated the entropy based on the probability of the test unit failing and further metrics regarding the duration of its execution. From the Git history, they calculated the number of changes to files of each file extension in the project in the past 3, 14, and 54 days. There were 38 different file extensions in the project, which combined with the three window sizes, led to 114 features. They also one-hot encoded the submodule each test unit belonged to, contributing nine boolean features. Furthermore, they measured the number of modified files in the current pull request and the number of distinct contributors to produce two additional numeric features. Following cross-validation and feature selection, their empirical evaluation demonstrated an F1 score of 95.5% using a gradient boosting classifier and a feature set consisting of the flip rate (with a reciprocal squared decay function), the number of changes to files with the `.cpp` extension in the past 54 days, and the number of modified files in the current pull request.

Concurrency

Dong et al. [32] presented FLAKESCANNER, a technique for detecting concurrency-related flaky tests in Android apps. All Android apps have a main or “UI” thread that is the only thread that had access to GUI elements. When a user performs an action such as tapping on an element, an “event” is enqueued on the UI thread and eventually dequeued by its event loop for processing. If this processing is going to take a long time, the UI thread will delegate it to a background thread. Once the background thread has finished, it will notify the UI thread by sending it an event. When testing Android apps, a testing thread sends events to the UI thread to simulate user input. The significant asynchronicity that can occur between the UI thread, the testing thread, and an arbitrary number of background threads, can lead to concurrency-related flaky tests. The key innovation behind FLAKESCANNER is to repeatedly execute test cases while reordering the events in the UI thread. The authors use dynamic analysis to ensure FLAKESCANNER does not schedule infeasible orders (scheduling an event before another event on which it is dependent). They evaluated their technique on 33 subject apps. They found that FLAKESCANNER was able to detect 45 out of 52 flaky tests known by the maintainers in 10 of the 33 apps. On average, it detected a flaky tests within three test runs. They also found that it outperformed SHAKER (see Section 2.5.1) and plain rerunning 100 times both in terms of the number of detected flaky tests and the average execution time. Finally, FLAKESCANNER detected 245 previously unknown flaky tests in 19 apps. The authors reported 20 of these and 13 were confirmed and addressed by the developers.

Vocabulary-Based Detection

Camara et al. [20] performed a replication study of Pinto et al. [128]. As explained in detail in Section 2.5.1, Pinto et al. performed an empirical evaluation on the performance of machine learning classifiers for detecting flaky tests based on identifiers in the test case code. This is often referred to as a “vocabulary-based approach” [6, 8]. As well as calculating various performance metrics for the classifiers, Pinto et al. also evaluated the information gain offered by individual tokens. In their replication, Camara et al. set out to investigate whether they could obtain similar results using different types of machine learning classifiers or different implementations of the same types of classifiers. To that end, they replicated the methodology of Pinto et al. with three additional classifiers and using `scikit-learn` instead of `Weka`. Their results were very similar to Pinto et al., with the change of implementation having only a small impact on the performance metrics

for each classifier and the three additional classifiers having similar performance to the original ones. Furthermore, they found that the tokens with the most information gain were associated with executing and coordinating tasks and persistence, similar to the original study. Camara et al. also reproduced the methodology of Pinto et al. but using different datasets to evaluate the performance of the classifiers. In the original study, Pinto et al. calculated performance metrics over the entire subject set of test cases with no consideration for project boundaries. Camara et al. found that the performance of all the classifiers was very poor when applied to projects outside of the training data, demonstrating a low degree of *inter-project generalisability*.

Test Smells

Camara et al. [19] presented an empirical study on the use of test smells for the prediction of flaky tests. Using a dataset of 24 projects containing 49,919 test cases, they evaluated the performance of eight types of machine learning classifiers, including random forest, logistic regression, and KNN. They encoded each test case with feature vectors indicating the presence of 19 possible test smells, augmented with two additional numerical features: the number of lines of code in the test case, and the total number of smells present. Included in the 19 smells were *assertion roulette*, when a test case has multiple assertions, *mystery guest*, when a test case instantiates file and database classes, and *sleepy test*, when a test case invokes the `Thread.sleep` method. Following the classic evaluation methodology of splitting the dataset into distinct training and testing sets, they observed an F1 score of 0.83 for the random forest and decision tree classifiers. Following the same inter-project methodology applied in their previous work [20], they observed considerably poorer performance, establishing once again that the classifiers did not generalise well to projects outside of their training data. Finally, they went on to compare the test small-based approach to the vocabulary-based approach used by Pinto et al. [128]. They found the vocabulary-based approach performed better following the classic evaluation methodology but significantly poorer following the inter-project methodology. This echoes their previous findings that this approach has very poor inter-project generalisability [20].

Order-Dependent Flaky Tests

Li et al. [99] introduced INCIDFLAKIES, a technique that builds upon IDFLAKIES [90], an approach for automatically detecting flaky tests in Java projects and classifying them as non-order-dependent (NOD) or order-dependent (OD) (see Section 2.5.2). The classification stage of IDFLAKIES involves repeatedly executing a whole test suite in random orders to identify OD flaky tests. The main contribution of INCIDFLAKIES is to reduce the execution time of IDFLAKIES when executed repeatedly over a project's development. Given the state of a project at one commit and the state at a later commit, INCIDFLAKIES seeks to detect any new OD flaky tests by only considering test cases affected by the changes between the two commits. To do so, it leverages two existing Regression Test Selection (RTS) techniques. For detecting OD flaky tests, it would not be enough to only rerun the test cases directly impacted by the changes, because to detect any OD flaky test requires the prior execution of other test cases to pollute the depended-upon shared state [141]. In addition to returning the list of test cases affected by recent changes, the leveraged RTS techniques also identify the production classes each test case depends on. From this list of classes, INCIDFLAKIES identifies all the additional test cases that need to be rerun alongside the test cases directly affected by the changes. These additional test cases represent the potential polluters. An empirical evaluation demonstrated that INCIDFLAKIES requires between 65% to 70% of the time taken by IDFLAKIES, depending on the choice of RTS technique.

Wei et al. [162] performed an empirical study on *Non-Idempotent-Outcome* (NIO) flaky tests. They defined NIO flaky tests as test cases that give a different outcome after repeated, consecutive execution within the same testing session. They also described them as being simultaneously *latent-victims* and *latent-polluters*. They defined a latent-victim as a test case that contains brittle assertions (see Section 2.3.2), such that it could be, or currently is, a victim test case (order-dependent, see Section 2.6.2). They defined a latent-polluter as a test case that modifies shared

state, such that it could, or currently does, cause a victim flaky test to fail. Therefore, the authors motivated their work by arguing that detecting NIO flaky tests is useful because they could preempt future test order dependencies. Their approach for detecting NIO flaky tests involved executing every test method twice with three possible levels of process-based isolation. They implemented their approach for both Java and Python programs and applied it to 127 Java modules and 1,006 Python projects used as subjects in previous studies on flakiness [59, 93, 163]. They detected a total of 223 NIO flaky tests in 34 of the Java modules and 138 NIO flaky tests in 90 of the Python modules. Given their dataset came from previous studies, many test cases were already labelled as victims or polluters. This enabled the authors to calculate the overlap with the detected NIO flaky tests. Of the 223 Java NIO flaky tests, 13 were victims but not polluters, 7 were polluters but not victims, and 8 were both victims and polluters. Of the 138 Python NIO flaky tests, 36 were victims (the Python dataset was not previously labelled for polluters).

Pre-Trained Language Models

Fatima et al. [41] presented FLAKIFY, a machine-learning based technique for detecting flaky tests based purely on their source code. To encode test cases, FLAKIFY uses CodeBERT, a pre-trained source code language model provided by Microsoft. Given up to 512 tokens, representing the source code of a test case, CodeBERT produces an integer-valued vector with 768 elements. For large test cases with more than 512 tokens, FLAKIFY only tokenises the parts of the test case that are likely to be relevant to flakiness. To do so, FLAKIFY parses the abstract syntax tree of the test case code to extract statements that match predefined patterns corresponding to various test smells. Given the vector representations of test cases, the actual classification is performed by a feed-forward neural network. To evaluate the technique, the authors performed an empirical evaluation using the same dataset as used by Alshammari et al. [8] in their evaluation of FLAKEFLAGGER (see Section 2.5.1). This was to enable a direct comparison with FLAKEFLAGGER. They performed both standard cross-validation and inter-project validation to evaluate the detection performance of FLAKIFY. In the latter case, the authors trained FLAKIFY using the test cases of every subject project excluding one and then evaluated it on that excluded project. Using cross-validation, FLAKIFY achieved an F1 score of 0.79 and FLAKEFLAGGER achieved 0.65 (see Table 2.19). Using inter-project validation, FLAKIFY achieved a mean F1 score of 0.73 and FLAKEFLAGGER achieved a mean of 0.07. These results led the authors to conclude that FLAKIFY was better suited at detecting flaky tests than FLAKEFLAGGER, not only because of the greater detection performance, but also because FLAKIFY does not require the execution of any test cases to produce features.

Qin et al. [133] proposed PEELER, another technique for detecting flaky tests without executing them. Unlike other static approaches, PEELER constructs a *Test Dependency Graph* (TDG) that captures the data dependency relationships between statements of the test case code and the code under test. From the TDG, PEELER enumerates every *contextual path*, that is a path from a start node to a node representing an assertion statement. From these contextual paths, PEELER leverages code2vec, a pre-trained language model for source code comparable to CodeBERT, to produce a vector encoding of the test case. To evaluate PEELER, the authors performed cross-validation using the same subject set as used by Alshammari et al. [8] to evaluate FLAKEFLAGGER. Their results showed that PEELER achieved an F1 score of 0.85 and FLAKEFLAGGER achieved 0.63 (see Table 2.19). They went on to evaluate their approach on three further projects that were not part of the subject set. To do so, they trained PEELER on the entire subject set and applied it to the three project in turn. In total, PEELER indicated that 65 test cases out of 1,835 were flaky. Of these 65, the authors reported the 21 test cases with a predicted probability of flakiness greater than 0.7 to the respective maintainers of the three projects. At the time of publication, 12 had been confirmed, 9 were still pending, and none had been refuted.

<p>Conclusion for RQ3: What insights and techniques can be applied to detect flaky tests? Three studies introduced automated techniques for detecting flaky tests based on data from a project’s CI and version control system. They are all based on machine learning</p>

Table 2.19: Precision (**Pr.**), recall (**Re.**), and F1 score (**F1**) of three machine learning-based flaky test detection techniques from three studies. These metrics are calculated from cross-validation on the same subject set of 24 open-source Java projects. Dashes indicate that a study did not evaluate a technique.

Technique	Alshammari et al. [8]			Fatima et al. [41]			Qin et al. [133]		
	Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1
FLAKEFLAGGER [8]	0.60	0.74	0.66	0.60	0.72	0.65	0.58	0.69	0.63
FLAKIFY [41]	-	-	-	0.70	0.90	0.79	-	-	-
PEELER [133]	-	-	-	-	-	-	0.77	0.85	0.81

classifiers such as KNN and tree-based ensemble models. Lampel et al. [94] introduced a technique that detects if a build failure is due to flaky tests or not. Ahmad et al. [2] introduced MDFLAKER to determine if a specific test failure is flaky. Finally, Gruber et al. [58] evaluated a range of classifiers to detect if a test case was flaky based on its history and recent changes to the project. Dong et al. [32] introduced FLAKESCANNER for detecting concurrency-based flaky tests in Android apps. An empirical evaluation involving 33 apps found that it was able to detect 45 out of 52 known flaky tests. Camara et al. [20] performed a replication of Pinto et al. [128], that previously examined the performance of machine learning classifiers for detecting flaky tests based on identifiers in the test case code. By evaluating the approach on test cases from projects outside the classifier’s training data, the authors demonstrated that it has poor inter-project generalisability. Camara et al. [19] also performed an empirical study on the use of test smells for the prediction of flaky tests with machine learning classifiers. Once again, they found that while the technique showed promise following a cross-validation evaluation methodology, it demonstrated poor generalisability when evaluated on projects outside of the training data. Two studies introduced techniques related to the detection of OD flaky tests. Li et al. [99] introduced INCIDFLAKIES, a technique that optimises IDFLAKIES [90] by leveraging RTS techniques to limit the number of test cases that need to be executed in random orders based on recent changes to the project. Following an empirical evaluation, they found that INDIDFLAKIES requires between 65% to 70% of the time taken by IDFLAKIES. Wei et al. [162] presented a study on NIO flaky tests, that have the potential to become OD flaky tests as the test suite evolves. Finally, two studies presented machine learning-based techniques based on pre-trained language models [41, 133]. The results of their empirical evaluations, as compared to previous work [8], are summarised in Table 2.19.

2.8.4 RQ4: What insights and techniques can be applied to mitigate or repair flaky tests?

Adjusting Assertion Bounds

Dutta et al. [36] presented FLEX, a technique for automatically fixing flaky tests due to randomness in machine learning algorithms. The technique repairs test cases that contain assertions comparing actual and expected values representing the quality of the outcomes of machine learning algorithms. For example, such an assertion could assert that the accuracy of a classification algorithm is greater than some upper-bound (e.g., 0.95). While this randomness could trivially be controlled by setting the seed of the random number generators used by such algorithms, developers may be hesitant to do so because this may limit the number of possible scenarios covered by the test case [34]. FLEX is based on Extreme Value Theory (EVT), a branch of statistics used to model extreme events. The authors use the Peak Over Threshold (POT) method from EVT to estimate the tail distribution of the quality value. With this method, the tail distribution converges in the limit to an instance of the Generalised Pareto Distribution (GPD), parameterised by a shape parameter that determines if the value has a left or right tail that exponentially bounded. If it is, the tail quickly converges to the GPD and can be used to estimate an appropriate bound

in the assertion. Otherwise, FLEX resorts to other fixing strategies, including simply annotating the test case as flaky as a last resort. Following an empirical evaluation involving 35 existing flaky tests in 21 projects, the authors found that FLEX was able to address 28, though in 9 instances the technique resorted to just annotating the test case.

Predicting Root Causes

Ahmad et al. [3] introduced a tool named FLAKYPY for detecting and identifying the root cause of flaky tests. The user of FLAKYPY specifies a test case and a number of reruns to perform. The tool then executes the test case that many times, and in each run, traces the test execution. This takes the form of a list of entries indicating one of three events: the execution of a line, a function call, or a return from a function. The entry for a function call also records the values of the local variables. After the test executions, FLAKYPY identifies *divergences* and *anomalies* in the execution traces. It inspects only the failing runs for divergences, which are points where the trace diverges. It defines an *anomaly* as a local variable which takes a value during a failing run that is not observed in any passing runs with a matching execution trace. When a failing run contains a divergence, only the anomalies in function calls that returned before the point of divergence are considered. The tool classifies anomalies as either *inconsistent-in-failing*, where the anomalies have different values across the failing runs, or *consistent-in-failing*, where they have matching values across all failing runs. FLAKYPY reports consistent-in-failing anomalies as a potential root cause of the flakiness.

Akli et al. [6] presented FLAKYCAT, a technique for predicting the root cause category of a flaky test. The technique is based on Few-Shot Learning (FSL), an approach in machine learning for training classifiers with imbalanced datasets where particular classes may have very few training examples. FSL is particularly suited to this application, since some flaky test categories are considerably rarer than others (see Table 2.5) and so it is difficult to acquire examples. Given a test case, FLAKYCAT tokenises the source code and leverages CodeBERT to produce a vector representation. This forms the input to a *Siamese network*, a type of neural network designed for FSL. To evaluate their technique, the authors combined existing flaky test datasets to produce a new dataset of 343 flaky tests. Each flaky test is associated with a label, indicating its root cause category out of ten possibilities. In this dataset, the most common four categories were asynchronous waiting, concurrency, time, and unordered collections. Following 4-fold stratified cross-validation, the weighted F1 scores with respect to predicting these four categories were 0.74, 0.46, 0.66, and 0.85 respectively.

Mitigating External Assumptions

Dietrich et al. [30] presented their approach for handling test cases with network dependencies. Initially, they described the assumption feature of JUnit. Similar to an assertion, it allows a developer to test a condition. Unlike an assertion, if the condition is false then the state of the test case becomes `skip`, indicating that it has been skipped. Unlike `fail`, this does not result in the entire test suite run failing. They also explained how the `error` test case state is distinct from the `fail` state. During test case execution, if an exception occurs that is not handled by the test case, the test case enters the `error` state. This can be thought of as the implicit assertion that no test case will throw exceptions (other than `AssertionError`). The authors remarked how, in the case of network-dependent test cases, it is difficult for developers to reliably include assumptions to check that the network is available. This is because, while it may be available when checking the assumption, it may become unavailable during the actual execution of the test case. This led them to introduce the notion of *external assumptions*, that are assumptions that cannot be checked using a unit testing API such as JUnit. They also introduce the concept of *sanitisers*, testing framework extensions that instrument test case execution and check for external assumptions on-the-fly. The authors implemented a sanitiser to handle the assumption that the network is always available. The sanitiser instruments the constructors of three specific exception classes associated with network availability issues. When an instance of one of these classes is

created during test case execution, a flag is set to communicate to the test case that a network exception has occurred. If the state of the test case ends up as `fail`, the sanitiser changes it to `skip` if the flag was set. If the state was `error`, it changes it to `skip` if the network-related exception is visible in the stack trace.

Developer Studies

Habchi et al. [63] interviewed 14 practitioners about their views on flaky tests. They set out to understand how developers address flaky tests and how mitigation measures could be improved with automated tools. They found that practitioners avoid flaky tests by implementing stable testing infrastructure, involving mock servers and containerisation, by defining good testing guidelines, such as “the testing pyramid” which discourages excessive integration and end-to-end test cases, and by limiting external dependencies. In terms of how flakiness mitigation could be improved with automated tools, the developers expressed the need for reliable root cause identification and reproduction. They also suggested that managing flaky tests would be either with effective tools for monitoring test case execution and analysing generated logs.

Similarly, Gruber et al. [56] aimed to understand the most common mitigation strategies applied by developers and their wishes for automated tools. They found that rerunning and rewriting test cases were by far the most common approaches, and automated techniques were rarely applied by developers. They summarised the main wishes of developers regarding automated tools as: dashboards to visualise test outcomes over time, IDE plugins to detect potentially flaky tests statically, and automated debugging and root cause analysis.

Open-Source Projects

Hashemi et al. [69] conducted an empirical study of flaky tests in JavaScript projects. Like Luo et al. [105], they categorised the repair strategies applied by developers to repair flakiness in 481 commits within the top-40 JavaScript projects on GitHub in terms of number of stars. They found that in 82% of cases the developers attempted to repair the flakiness. In these cases, the results were broadly similar to that of Luo et al. and Eck et al. [37] (see Table 2.12), with adding explicit waits for events, introducing/extending delays, and reordering code as common changes applied to flaky tests of the concurrency and asynchronous wait categories. Of the remaining commits, 7% aimed to reduce the flakiness of test cases without completely removing it, 7% skipped or disabled the test case, 2% quarantined the test case for a later fix and 2% deleted the test case.

Reproducing Concurrency Flakiness

Leesatapornwongsa et al. [96] presented FLAKEREPRO, a technique for reproducing concurrency-related flaky tests. Given a known flaky test and the error message from the failure, FLAKEREPRO explores the space of potential thread interleavings. Because this space grows exponentially with the number of threads, FLAKEREPRO follows heuristics to make the exploration tractable. Following data- and control-flow analysis, FLAKEREPRO only considers interleavings of shared memory accesses that can influence the target error message. The authors remarked that many concurrency-related flaky tests follow the asynchronous waiting pattern [105], where the test case fails because an object involved in an assertion reads a critical shared value in the absence of a write. Therefore, in its exploration of possible interleavings, FLAKEREPRO prioritises patterns of critical access instance pairs where the reading object is fixed to be the instance from the failed assertion. Another heuristic it applies is to deprioritise exploration of repeated critical accesses in a loop after the first case. The technique uses binary-level instrumentation to induce different interleavings during exploration and to reproduce a specific interleaving that induces the given error message once found. An empirical evaluation involving 22 projects internal to Microsoft found that FLAKEREPRO could deterministically reproduce 26 out of 31 flaky tests after exploring fewer than six interleavings on average.

Order-Dependent Flaky Tests

Li et al. [98] presented ODREPAIR, a tool for generating cleaners for victims. The tool builds upon IFIXFLAKIES [141] (see Section 2.6.2) that automatically detects victims and their cleaners (test cases that reset the shared state that enable the victim to pass). Given a victim's cleaners, IFIXFLAKIES uses delta-debugging to identify the necessary statements in the cleaners to produce a patch to repair the victim. One limitation of IFIXFLAKIES is that if no cleaners exist in the test suite, it cannot repair the victim. ODREPAIR addresses this gap by generating a cleaner for a given victim so that IFIXFLAKIES can repair it. Given a victim and its polluter (the test case that corrupts the shared state causing the victim to fail), ODREPAIR identifies the polluted state (in-memory heap state reachable from static fields) and searches through the code base for methods that can reset the shared state. ODREPAIR then leverages the automatic test generation tool Randoop to generate tests for each reset method. The goal of this is to produce cleaners for the victim, rather than genuine test cases for finding bugs. Once ODREPAIR has generated cleaners, it passes them IFIXFLAKIES to verify that they are actually cleaners for the victim and to then generate a patch. An empirical evaluation involving a subject set of 327 victim flaky tests demonstrated that ODREPAIR can automatically identify the polluted state for 181. Of these, ODREPAIR generated a patch for 141, which was 24 more than IFIXFLAKIES alone.

Localizing Flakiness

Habchi et al. [64] performed an empirical study into the applicability of Spectrum-Based Fault Localisation (SBFL) for identifying classes in the code under test that may be responsible for flakiness. In SBFL, a *suspiciousness score* is calculated for every program element (e.g., line, function, class, etc.) based on a formula concerning the number of passing and failing test cases that cover it, and the number of passing and failing test cases that do *not* cover it. A high suspiciousness score suggests that a particular program element may contain a bug. In this study, the authors focused on classes and substituted non-flaky and flaky test cases for passing and failing test cases in the formula. As their dataset, they collected 38 commits that repaired flaky tests, involving changes to the code under test, in five open-source Java projects. For each commit, the authors inspected the diff and message to manually label the flaky tests, the “flaky classes” (modified classes in the code under test assumed to be responsible for the flakiness), and the root cause category of the flakiness. They also executed the test suites to collect coverage information. They initially evaluated four well-established SBFL formulae and an approach based on Genetic Programming (GP) that evolves a formulae. They found that, using the four existing formula, the flaky classes were in the top-10 in terms of suspiciousness score for between 47% and 53% of the commits. The GP approach performed the best at 58% in the top-10. They repeated this methodology but for just the commits in each of the root cause flakiness categories. They found concurrency and asynchronous waiting to be the most common categories, accounting for 16 and 10 commits respectively. With respect to concurrency, they found that with the GP approach the flaky classes were in the top-10 for 50% of the commits. For asynchronous waiting, it was 80%.

Gruber et al. [57] performed a related study but focused on statements rather than classes and applied a more nuanced approach for adapting SBFL for localizing flakiness. While still substituting non-flaky and flaky test cases for passing and failing test cases in the SBFL formulae, they considered a statement to have been covered by a flaky test only if it was covered in every run of the test suite following repeated executions to measure coverage. In contrast, they considered a statement to have been covered by a non-flaky test if it was covered in any run. They referred to this approach as Spectrum-based Flaky Fault Localisation (SFFL). Their motivation for this was based on their observation that 80% of all flaky tests and 12% of all non-flaky tests had at least two different coverage patterns, and so to apply standard SBFL definitions as Habchi et al. [64] did could lead to misleading suspiciousness scores. For their evaluation, Gruber et al. executed the test suites from 1,006 Python projects 800 times and identified 874 NOD and 4,633 OD flaky tests. Of these, they randomly sampled 89 NOD and 29 OD to manually label with fault locations to form their ground-truth, against which they evaluated five existing SBFL formulae adapted

for SFFL. As an evaluation metric, they used EXAM score. For a given statement, the EXAM score is the ratio of its rank in the suspiciousness score ratings over the total number of lines in the project. This is a value between zero and one where a lower score is better. Using the DStar formula, the authors observed an average-case mean EXAM score of 0.035.

Conclusion for RQ4: What insights and techniques can be applied to mitigate or repair flaky tests?

Dutta et al. [36] presented FLEX, an automated technique that adjusts assertion bounds in flaky tests caused by algorithmic randomness. An empirical evaluation involving 35 existing flaky tests showed that FLEX was able to address 28 of them. Two studies presented automated techniques to predict the root causes of flaky tests. Ahmad et al. [3] presented FLAKYPY that compares the execution traces between passing and failing runs of flaky tests to identify divergences and pinpoint the possible code locations where the flakiness originates from. Akli et al. [6] presented FLAKYCAT that leverages a Siamese neural network to detect the root cause category of a flaky test. Leesatapornwongsa et al. [96] presented FLAKEREPRO for reproducing the specific thread interleavings that cause concurrency-related flaky tests to fail. An empirical evaluation involving 22 projects internal to Microsoft found that FLAKEREPRO could deterministically reproduce 26 out of 31 flaky tests. Li et al. [98] presented ODREPAIR, a technique that leverages an existing automated test generation tool to produce cleaners for victim flaky tests when they do not currently exist in the test suite. This is to enable IFIXFLAKIES [141] to generate a patch for the victim. An empirical evaluation involving 327 victim flaky tests demonstrated that ODREPAIR was able to repair 24 more than IFIXFLAKIES. Two studies investigated the applicability of existing SBFL techniques for identifying elements of the code under test that may be responsible for flaky tests. Habchi et al. [63] focused on classes and adapted existing formulae by substituting non-flaky and flaky tests for passing and failing test cases. Gruber et al. [57] focused on line abd proposed a more nuanced approach for adapting SBFL to debugging flaky tests.

2.9 Conclusion

In this survey, I examined 76 sources related to test flakiness and answered four research questions regarding their origins, consequences, detection, mitigation and repair. My first research question considered the causes of flaky tests and the factors associated with their occurrence. I identified a taxonomy of general causes and considered the specific factors associated with order-dependent flaky tests in particular as well as how the causes of flakiness can differ across specific types of projects. I then set out to examine what areas of testing are impacted by flaky tests, allowing me to answer my second research question regarding their consequences and costs. I found that they impose negative effects on the reliability and efficiency of testing in general, as well as being a hindrance to a host of techniques in software engineering. My third research question concerned the detection of flaky tests. I first examined approaches for detecting flaky tests in general, before discussing a series of tools for the detection of order-dependent flaky tests in particular. My final research question set out to investigate the ways in which the negative impacts of flaky tests can be mitigated, or going a step further, repaired entirely. To that end, I presented a range of techniques for alleviating some of the problems presented by flaky tests and described a technique for the automatic repair of order-dependent tests. Overall, I provided researchers with a snapshot of the current state of research in the area of flaky tests and identified topics for which further work may be beneficial. I also gave those wishing to familiarise themselves with the field the necessary reading for a succinct knowledge of the insights and achievements to date.

In pursuit of the first goal of this thesis, to further the understanding of flaky tests, this chapter has examined the problem in great detail through the lens of published research. In spite of the scientific rigour of the papers I reviewed, there is a danger that their findings and insights do not fully reflect, or are not fully applicable to, the day-to-day realities of professional software developers dealing with flaky tests. While previous studies have directly consulted developers [37, 71], many more are based on the authors' observations following automated experiments.

Therefore, a survey of developers' experiences of flaky tests, guided by the findings from the literature, is required for a more well-rounded understanding of flaky tests. This is essential to keep the focus of the research community on issues that are most likely to benefit industry.

Chapter 3

Surveying the Developer Experience of Flaky Tests

The contents of this chapter is based on “O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 253–262, 2022”.

3.1 Introduction

Since flaky tests are primarily a developer problem, there is an underutilised opportunity to acquire valuable insights from those who experience them first-hand. Where previous studies do exist, they focus on specific organisations and developers’ self-reported experiences [37, 71], two potential sources of bias [31]. In this study, I examine multiple sources to understand how developers define and react to flaky tests and their experiences of the impacts and causes. I use the findings from the literature discussed in Chapter 2, in combination with insights from grey literature, to inform the design of a developer survey. I received 170 responses after deploying the survey on social media with the help of my collaborators.

In addition to the survey, I searched StackOverflow and filtered the results to produce a dataset of 38 threads about flaky tests. With the help of my collaborators, I performed thematic analysis [29] on the questions and accepted answers to gain insights into the flaky tests that developers require assistance to diagnose and repair. Through this unique perspective, I was able to identify themes regarding additional causes and actions that were not revealed by the developer survey.

By examining flaky tests through the lens of developers, as I did through the lens of published research in Chapter 2, I am able to further the understanding of flaky tests through multiple, occasionally conflicting, perspectives. As such, some of the findings of this chapter support previous literature while others were unanticipated. For example, I found that participants who experience flaky tests more often may be more likely to ignore potentially genuine test failures, supporting the position of experts such as Martin Fowler [180]. I also found that participants rated asynchronicity and concurrency as only the fourth and fifth most common causes of flaky tests respectively, despite studies reporting these to be the most common [37, 105, 136]. I make all data and artifacts available in the replication package [194].

In summary, the main contributions of this study are as follows:

1. **Developer Survey (Section 3.2.1)**: I designed a survey based on previous literature and received 170 responses. Through numerical and thematic analysis, I identified alternative definitions of flaky tests, the most significant impacts of flaky tests, the most frequent causes of flaky tests, and the most common actions developers perform in response to flaky tests.
2. **StackOverflow Threads (Section 3.2.2)**: I procured a dataset of 38 StackOverflow threads and through thematic analysis I offer a unique insight into the causes of flaky tests

experienced by developers and the strategies that they suggest to repair them, independent of what they self-reported in the developer survey.

3. **Findings and Recommendations (Section 3.4):** I surfaced a range of findings that support previous literature and some that were more unforeseen. From these, I offer actionable recommendations for both software developers and researchers.

3.2 Methodology

To understand how software developers define, experience, and approach flaky tests, I asked the following research questions:

RQ1: Definition: *How do developers define flaky tests?*

RQ2: Impacts: *What impacts do flaky tests have on developers?*

RQ3: Causes: *What causes the flaky tests experienced by developers?*

RQ4: Actions: *What actions do developers take against flaky tests?*

To answer these, I conducted a multi-source study consisting of a developer survey, with both closed- and open-ended questions, and an analysis of StackOverflow threads. I performed numerical analysis on the closed-ended survey questions and thematic analysis [29] on the open-ended questions and the StackOverflow threads. Before conducting the study, I received ethical approval for the developer survey from the University of Sheffield (see Appendix A). I include the participant information sheet in the replication package [194].

3.2.1 Developer Survey

I designed a survey of 11 questions for software developers. Some presented a list of prepared statements and asked participants to respond to them in a closed-ended fashion. I reviewed the findings from Chapter 2 and other grey literature to ensure the relevance of these statements. With the help of my collaborators, I disseminated the survey on Twitter and LinkedIn, specifically asking for developer participants. We also circulated the survey among Sheffield Digital, a regional technology business forum [276]. The 11 survey questions are as follows:

SQ1: *A flaky test is a test case that can both pass and fail without any changes to the code under test. Do you agree with this definition?* Participants could indicate that they agreed or disagreed. This is a common definition, though it is not universal. Vahabzadeh et al. [155] only considered test cases whose non-determinism was caused by timing or concurrency to be flaky tests. On the other hand, Shi et al. [139] included test cases with inconsistent coverage in their study on mitigating the effects of flaky tests on mutation testing.

SQ2: *If you answered “No, I do not agree” to the previous question, please give your own definition of a flaky test.* This gave participants who disagreed with the proposed definition the opportunity to offer their own. Together with **SQ1**, these questions address **RQ1**.

SQ3: *How often do you observe flaky tests in the projects you’re currently working on?* Participants could answer *Never*, *A few times a year*, *Monthly*, *Weekly*, or *Daily*. I used this to gauge the prevalence of flaky tests and as a demographic variable in the analysis.

SQ4: *To what extent do you agree with the following statements:* To address **RQ2**, this question posed eight statements and asked participants to rate their agreement on a four-point scale. With reference to previous literature reviewed in Chapter 2, the statements are as follows:

SQ4.1: Reliability: *Flaky tests reduce the reliability of testing.* Vahabzadeh et al. [155] categorised 443 bug reports regarding test cases. They found that 97% caused the test to fail without indicating a bug, i.e., they were *false alarms*. They categorised 53% of these as either flaky tests, resource mishandling, or caused by factors in the execution environment. According to the proposed definition in **SQ1**, I also consider the latter two categories as flaky tests.

SQ4.2: Efficiency: *Flaky tests reduce the efficiency of testing.* A specific category of flaky tests, known as *order-dependent* (OD) flaky tests [90, 141, 175], are influenced by previously executed

test cases. Techniques to improve the efficiency of testing by reordering, reducing or splitting-up the test suite are unsound when OD tests are present. For instance, in the test suites of 11 Java modules, Lam et al. [91] found that 23% of OD tests failed after they applied test case prioritisation, 24% after test case selection, and 5% after parallelisation.

SQ4.3: Productivity: *Flaky tests lead to a loss of productivity.* John Micco [183] explained that developer productivity relies on the capability of test cases to identify issues in a timely and reliable manner. Flaky tests are unreliable and could harm the productivity of the developers who experience them as well as those who rely on the productivity of those developers [190].

SQ4.4: Confidence: *Flaky tests lead to a loss of confidence in testing.* Given how flaky tests may manifest as a false alarm [155], there is a danger that developers will lose confidence in testing if they continuously experience flaky tests. This could lead to a culture of ignoring tests, which may cause genuine bugs to go unnoticed [181].

SQ4.5: CI: *Flaky tests hinder continuous integration (CI).* Durieux et al. [33] analysed over 75 million build logs on Travis CI. They found that 47% of previously failing builds that were manually restarted by a developer subsequently passed. Since no changes were involved, these builds may have failed due to flaky tests.

SQ4.6: Ignore: *Flaky tests make it more likely for you to ignore (potentially genuine) test failures.* Martin Fowler [180] explained how developers may be tempted to ignore flaky test failures. He explained that if a test suite contains too many flaky tests, developers could lose the discipline to ignore just the flaky failures. Rahman et al. [134] found that ignoring test failures, flaky or not, was associated with a higher volume of crashes due to missed bugs.

SQ4.7: Reproduce: *It is difficult to reproduce a flaky test failure.* Lam et al. [92] described how, upon encountering a failing test, a developer might rerun it in isolation from the rest of the test suite in order to reproduce the failure and debug the code under test. They reran in isolation, for 4,000 times each, the 107 flaky tests from 26 Java modules, only reproducing the failures of 57 and concluding that this may be ineffective at reproducing flaky test failures.

SQ4.8: Differentiate: *It is difficult to differentiate between a test failure due to a genuine bug and a test failure due to flakiness.* Lamyaa Eloussi [185] remarked how flaky tests are a source of wasted time, particularly during regression testing, where flaky test failures may appear linked with a commit but can actually be unrelated.

SQ5: *In the projects you're currently working on, how often have you encountered flaky tests caused by...* I gave participants a list of causes and asked them to rate on a four-point scale how often they had experienced them. The list of causes is as follows:

SQ5.1: Waiting: *Not correctly waiting for the results of asynchronous calls to become available.* This has been considered in numerous studies and is widely agreed upon by researchers to be a leading cause of flaky tests [37, 89, 105, 136]. For example, a flaky test that spawns a new process to perform an operation but does not wait for the process to finish falls under this category.

SQ5.2: Concurrency: *Synchronisation issues between multiple threads interacting in an unsafe or unanticipated manner (e.g., data races, atomicity violations, and deadlocks).* Like **SQ5.1**, studies point to this category as being very common. Eck et al. [37] explained that flaky tests caused by *local* thread synchronisation issues belong in this category, while synchronisation issues with *remote* resources, such as web servers or external processes, would be **SQ5.1**.

SQ5.3: Setup/teardown: *Tests not properly cleaning up after themselves or failing to set up their necessary preconditions.* Many studies identified OD flaky tests to be a very prevalent category [37, 59, 90, 105]. Bell et al. [14] suggested that one cause may be the burden of writing correct setup and teardown methods, executed by a test runner before and after the main body of a test case. Shi et al. [141] differentiated between *victims*, an OD flaky test that fails if executed after a *polluter* test case, and *brittles*, an OD flaky test that only passes if executed before a *state-setter*. In the former, the victim does not perform proper setup and/or the polluter does not perform proper teardown. The latter instance is similar but reversed.

SQ5.4: Resources: *Improper management of resources (e.g., not closing a file or not deallocating memory).* The specific case of failing to release acquired resources (i.e., a resource leak) has been identified at a generally lower prevalence than the preceding categories in previous research

[37, 59, 89, 105]. Bearing some similarities to **SQ5.3**, the test case that improperly manages the resource may not be the test case that is flaky, but rather a subsequently executed test.

SQ5.5: Network: *Dependency on a network connection.* Any test case that requires a network connection will inevitably be flaky since infrastructure issues or periods of high traffic may cause the test case to fail. Several empirical studies have described this particular cause, with varying degrees of prevalence [37, 89, 105, 152].

SQ5.6: Random: *Not accounting for all the possible outcomes of random data generators or code that uses them.* Test cases that use random data, or cover code that utilises randomisation, can become flaky for a variety of reasons. One reason is that it may be difficult for developers to approximate test oracles, such as the appropriate range of output values in assertion statements [37, 116]. This is a particular problem for machine learning applications [35, 36].

SQ5.7: Time/date: *Reliance on the local system time/date.* Test cases that depend on time and date are fraught with difficulty, such as inconsistencies in representation and precision across systems as well as timezone conversion issues [37, 59, 89, 105].

SQ5.8: Floating point: *Inaccuracies when performing floating point operations.* Given their limited precision and other idiosyncrasies, floating point comparisons can sometimes produce unexpected results. In the context of flaky tests, previous work generally considered this specific cause to be quite rare [37, 89, 105].

SQ5.9: Unordered: *Assuming a particular iteration order for an unordered collection-type object (e.g., sets).* This is a special case of a general cause pertaining to assumptions regarding the implementations of non-deterministic program specifications [61, 140, 174].

SQ5.10: Unknown: *Reasons that cannot be precisely determined.* Finally, developers could indicate how often they had encountered flaky tests whose cause they could not precisely identify.

SQ6: *Have you encountered any other causes of flaky tests that we have not described above?* This question gave participants the chance to tell me about any other causes of flaky tests that I did not list in **SQ5**. Together with **SQ5**, these questions address **RQ3**.

SQ7: *After identifying a flaky test, how often do you...* This question offered a list of actions and participants could rate how often they perform them on a four-point scale. They were:

SQ7.1: No action: *Take no action.* Quite simply, a developer may choose to take no action when encountering a flaky test.

SQ7.2: Re-run: *Re-run the build.* Perhaps the most straight-forward action, a developer may just restart the failing build and hope that the flaky test passes this time. In their study of Travis CI build logs, Durieux et al. [33] found this to be a common practice.

SQ7.3: Document: *Document and defer (e.g., submit an issue/bug report).* A developer may not have the time to immediately repair a flaky test and may choose to document it for attention later. For example, they could raise an issue in a GitHub repository.

SQ7.4: Delete: *Delete the test.* Another straight-forward action is to permanently remove the flaky test from the test suite. Recounting on his experiences at Facebook, Kent Beck remarked how it was routine to delete non-deterministic test cases [191].

SQ7.5: Quarantine: *Quarantine the test.* Martin Fowler [180] advised that flaky tests should be *quarantined* from the main test suite into a dedicated test suite that is understood by the development team to be unreliable. He advised that developers should keep the quarantined test suite small by promptly fixing flaky tests. Otherwise, there is a danger of flaky tests being forgotten about and the whole process becoming equivalent to just deleting test cases.

SQ7.6: Mark skip: *Mark the test to be skipped or as an expected failure (e.g., xfail).* Many testing frameworks allow test cases to be marked as *skipped*, meaning they are not deleted from the test suite but are not executed either. Alternatively, some frameworks, such as `pytest`, allow test cases to be marked as *expected failures* or *xfails*. This signals to the testing framework that they are expected to fail, in which case they should not fail the entire test suite.

SQ7.7: Mark re-run: *Mark the test to be automatically repeated (e.g., by using the flaky plugin for pytest).* Often via the support of plugins, some testing frameworks allow test cases to be marked such that they are repeated some number of times upon failure. One example of such a plugin is `flaky` for `pytest` [206].

SQ7.8: Repair: *Attempt to repair the flakiness.* Finally, a developer may attempt to repair the underlying cause of the flaky test rather than just mitigating it with one of the previous actions.

SQ8: *Are there any other actions that you would take that we have not listed above?* In this question, participants could report any additional actions. This addresses **RQ4** along with **SQ7**.

SQ9: *Which languages are you currently developing in?* Participants could select from a list of programming languages, with an option to specify any other languages that I did not include. I asked this question to obtain further demographic information about the participant population.

SQ10: *How many years experience do you have in commercial and/or open-source software development?* Participants could select one of *0-1*, *2-4*, *5-8*, *9-12*, or *13+* years. Like **SQ3**, I used this as an additional demographic variable in the analysis.

SQ11: *Is there anything else that you would like to tell us about flaky tests?* The final question gave participants the opportunity to relay any miscellaneous insights they had about flaky tests.

3.2.2 StackOverflow Threads

I analysed StackOverflow threads where a developer asked for help addressing one or more flaky tests. I selected StackOverflow specifically due to its widespread popularity and its use in previous software engineering research [115]. This analysis adds further depth to the answer for **RQ3** by considering the causes of flaky tests that developers ask for help with, as opposed to the causes they report as the most common. It is also free of any self-reporting bias that may result from the survey [31]. For **RQ4**, this analysis offers insights into *how* developers repair flaky test cases, since the survey only asks participants *how often* they attempt to do so.

A thread on StackOverflow consists of a single question followed by answers. A user can indicate that an answer has addressed their question by *accepting* it. To find relevant threads, I used the website’s search feature. The query I used was “**flaky test hasaccepted:yes**”. The latter part of the query is a search operator that only matches threads where the user who asked the question has accepted an answer. This is to increase the probability that I can identify a recommended course of action for **RQ4**. From the search results, I created a dataset of relevant threads. I only included threads where the question was specifically asking about the cause of one or more flaky tests and/or how to repair them. I excluded threads where the question was more tangential, such as asking how to handle flaky tests in a specific testing framework [184]. This is because such threads do not present causes or possible repairs and are therefore of no use for addressing **RQ3** or **RQ4**.

3.2.3 Analysis

I designed **SQ4**, **SQ5**, and **SQ7** to be answered using a four-point Likert scale, quantifying agreement in the case of **SQ4** and frequency for **SQ5** and **SQ7**. Each of these questions asked participants to respond to a list of prepared statements. For each statement, I assigned a score between 0 and 3 to each point on the Likert scale. As an example, for each statement of **SQ4**, participants could select *Strongly disagree*, *Agree*, *Disagree*, or *Strongly disagree*, corresponding to a score of 0, 1, 2, or 3 for that statement, respectively. For each question, I calculated the mean score of each statement across all participants and four specific populations. The first two populations were participants who said they experienced flaky tests on at least a monthly basis and those who experienced them less frequently (see **SQ3**). I chose to split the participants by this criterion since those who frequently experience flaky tests may have different views to those who experience them rarely [180].

The final two populations were participants who said they had at least 13 years of software development experience and those who had fewer (see **SQ10**). I made this split because, when compared to participants with less experience, those with more may be better at accurately diagnosing the causes of flaky tests and may be more likely to take certain actions to address them. I excluded any respondent from the analysis of a particular question if they did not respond to all of that question’s statements. For each question, I calculated the ranks of every statement, based

on mean score, to quantify the most significant impacts in the case of **SQ4** and the most common causes and actions for **SQ5** and **SQ7**, respectively.

With the help of my collaborators, I performed *inductive thematic analysis* [29] on the responses to the open-ended survey questions (**SQ2**, **SQ6**, **SQ8**, and **SQ11**) and the StackOverflow threads. For each survey question, I met with my three collaborators (the three co-authors of the conference paper this chapter is based on [122]) and discussed each response. We split responses containing logical connectives such as “and” and “or” into their atomic components. We then assigned one or more labels or *codes* to each response, representing its key concepts. By collaboratively performing the coding, we minimised the impact of any individual biases and ensured our coding was as consistent as possible. From these codes we derived a set of themes, representing the definition of flaky tests, their causes, and developers’ actions against them for **SQ2**, **SQ6** and **SQ8**, respectively. For **SQ11**, the themes represent more general insights. We performed a very similar procedure for the StackOverflow threads. Next, we assigned themes regarding causes and actions to each thread in two separate sessions. Finally, we encountered several accepted answers prescribing multiple actions, in which case we assigned them to multiple action themes.

3.2.4 Threats to Validity

All methodologies carry the risk of biasing results, including this study’s. This section discusses both these risks and the mitigations.

Replicability: *Can others replicate the results?* The numerical analysis is straightforward to replicate. I make the response data from the developer survey and the Python script for performing the numerical analysis available as part of the replication package [194]. In general, thematic analysis is more challenging to replicate. Nonetheless, I include in the replication package the spreadsheets I used to facilitate the collaborative thematic analysis.

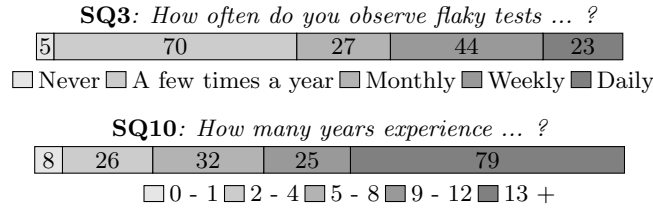
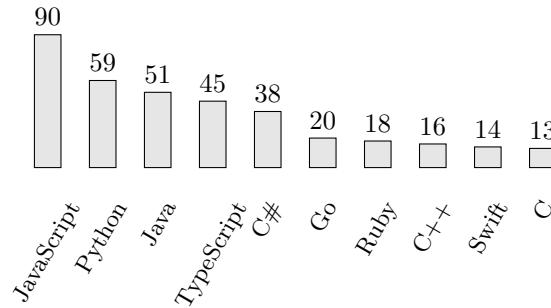
Construct: *Am I asking the right questions?* The construction of the study can bias the results, as in any study. To attain the highest quality of results possible, I designed a multi-source approach. The numerical analysis of the closed-ended survey questions provides broad, high-level insights. The thematic analysis of the open-ended questions offers a more specific, but much more detailed, understanding of developers’ experiences. Finally, the analysis of the StackOverflow threads provides an alternative perspective, free of any potential self-reporting bias [31]. I selected these three components to collect inherently different but complementary data, thereby giving a more complete understanding of flaky tests.

Internal: *Did I skew the accuracy of the results with how I collected and analysed information?* It is possible for the results of surveys to be impacted by biases, from both participants and researchers. During the numerical analysis, there is little I can do to mitigate this on the participants’ side, though since the analysis is purely mathematical in nature, there is very limited scope for researcher bias. During the thematic analysis, I mitigated any individual researcher bias by collaboratively performing all analyses. Due to the nature of the recruitment, I could not verify that participants genuinely were software developers. Therefore, I have no guarantee that the participant population accurately reflects the target population. I mitigated this by specifically asking for developers in the Twitter and LinkedIn posts and by making it clear in the participant information sheet that I was seeking developers. Furthermore, the technical nature of the questions mean the survey would be difficult for non-developers to complete.

External: *Do the results generalise?* I cannot make any claims regarding the generalisation of the results beyond the survey population. Even though this is a natural limitation of any survey-based study, I mitigated it by not targeting any specific organisation and recruiting participants through more than one platform.

3.3 Results

I received 170 responses to the survey. Figure 3.1 shows the results of **SQ3** and **SQ10**. For **SQ3**, just over half of the participants reported that they observe flaky tests on at least a monthly

Figure 3.1: Results for **SQ3** and **SQ10**.Figure 3.2: Results for **SQ9**: Which languages are you currently developing in? (Top 10 languages.)

basis. This shows that flaky tests are a frequent phenomenon, especially given that 23 reported experiencing them daily. For **SQ10**, just under half said they had 13 years or more of software development experience. For **SQ9**, the top three programming languages were JavaScript, Python, and Java. The distribution roughly corresponds to the most popular languages according to the 2021 StackOverflow developer survey [193] (see Figure 3.2). This reassures me that the participant population is generally representative of the wider community of developers.

After performing the search on StackOverflow, I found 169 threads. I carefully examined each one and narrowed them down to the 38 that are relevant according to the criteria in Section 3.2.2. I now answer each research question using the results of the analysis of the responses to the survey and the StackOverflow threads. See Table 3.1 for the results of the thematic analysis for **SQ11**.

3.3.1 RQ1: Definition

Of the 169 respondents who answered **SQ1**, 6.5% disagreed with the proposed definition. In order of prevalence, the themes following the thematic analysis for **SQ2**:

SQ2t1: Beyond code: *The definition extends beyond the test case code and the code that it covers.* Participant 97 (P97) said “... a flaky test is any test that changes from pass to fail (or vice versa) in different environments”. P147 relayed a similar view, but specifically for test cases that only fail in a CI environment. P27 stated more generally that a test case whose outcome depends on changes *irrelevant* to the code under test is flaky. Arguably, this includes the environmental changes referenced by P97 and P147 and more.

SQ2t2: Flaky code under test: *A flaky test can indicate that the code under test is flawed, rather than the test case itself.* In the words of P155, “... a flaky test is therefore either unreliable itself or it proves the code under test is flawed and unreliable”. P25 indicated that the term *flaky* is inappropriately used to blame test cases when their flakiness is inevitable if they test nondeterministic code.

SQ2t3: Beyond test outcomes: *A test case can be considered flaky despite having a consistent outcome.* P58 wrote “... this includes pass/fail, but can encompass other aspects such as coverage or test time”. P25 generalised by considering more abstract characteristics such as the extent that the test case controls the system under test.

Table 3.1: Themes for **SQ11** regarding miscellaneous insights about flaky tests, in order of prevalence.

Title and description	Representative quote
SQ11t1: Developer culture: The relationship between flaky tests and testing practices and developer culture.	“It’s often an organisational problem...” – P89
SQ11t2: Emotive response: An expression of anger or other emotion.	“They suck.” – P91
SQ11t3: Poor tooling support: Tooling for handling flaky tests is inadequate or not well known.	“Library support for automatically handling them in Scala is poor or not well popularised.” – P7
SQ11t4: Execution environment: The interplay between execution environment and flaky tests.	“Caused by poor quality of coding and poor test specification coupled with a lack of understanding of the environment.” – P101
SQ11t5: Silver lining: Flaky tests may have some utility.	“Flaky tests can be valuable as they often point to an underlying weakness in the codebase.” – P133
SQ11t6: Time/date logic: Test cases handling time/date logic are notoriously flaky.	“90% of the time it’s date and or timezone logic ...” – P28
SQ11t7: External service: Flaky tests caused by third party services.	“Recently, seen a lot of flaky tests when running CI on Azure due to failures to download libraries ...” – P31
SQ11t8: Not worth fixing: The resources required to repair flaky tests are too great to make it worthwhile.	“... We haven’t got the time to address them all.” – P64

SQ2t4: Learn to live with it: *Flakiness is an inevitable aspect of testing.* P62 agreed with the proposed definition, but indicated that some test cases may be flaky by nature, saying “... not all tests are deterministic”. P25 expressed that there may be limited value in labelling test cases as flaky, since they are an inescapable aspect of testing. Conceivably, **SQ11t5** is a continuation of this concept and indicates that some participants consider them to have value.

SQ2t5: Usefulness of the test: *A test case that cannot effectively identify bugs is flaky.* In reference to the proposed definition, P116 stated “I think it’s broader than that and includes things like tests that pass independent of conditions”. P101 said that a test case is flaky if it cannot clearly identify problems in the code under test. This theme is similar to **SQ2t3** but leans more towards bug-finding capability.

Conclusion for RQ1: Definition: *How do developers define flaky tests?* Most participants (93.5%) agreed with the proposed definition of flaky tests in **SQ1**. Following the thematic analysis for **SQ2**, I identified more general definitions. Some participants indicated that the definition should consider factors beyond the test case code or the code under test. Others expressed that only taking the outcome of a test case into consideration when defining flaky tests is not enough. They conveyed that other behaviours, such as coverage, and more abstract properties, such as usefulness, should be part of the definition. Several offered more digressive insights, such as test cases should not always be considered at fault for the flakiness, as it is an inevitable aspect of testing.

3.3.2 RQ2: Impacts

The top third of Table 3.2 shows the mean scores and ranks of each impact statement. For all participants, **SQ4.5** scored the highest. This indicates that developers strongly agree with the notion that flaky tests hinder CI. Second and third were **SQ4.3** and **SQ4.2**, regarding losses to productivity and the efficiency of testing, respectively. The lowest scoring impact was **SQ4.8**, which, as illustrated by the distribution bar, was the most evenly split between agreement and disagreement. This suggests that differentiating between a true test failure and a spurious failure

due to flakiness is relatively straightforward for some developers.

The most significant difference in mean score between participants who experienced flaky tests on at least a monthly basis and those who did not was for **SQ4.6**. This indicates that developers who experience flaky tests more often could be more likely to ignore potentially genuine test failures. Beyond that, the scores are similar, with both scoring **SQ4.5** the highest and **SQ4.8** the lowest. The scores between the participants with at least 13 years of development experience and those with fewer are also similar. In **SQ11t2**, participants expressed anger or frustration at flaky tests. P96 said they “they’re very annoying”. Along with **SQ4.4** and **SQ4.6**, this further evidences the psychological cost of flaky tests.

Conclusion for RQ2: Impacts: *What impacts do flaky tests have on developers?* The analysis for **SQ4** indicates that developers strongly agree with the notion that flaky tests hinder CI. They also agree that flaky tests lead to both a loss of productivity and a reduction in testing efficiency. Respondents were mixed with regards to the difficulty of differentiating between a failure due to a true bug and one due to flakiness, implying that some developers may not find this challenging. The analysis also suggests that developers who experience flaky tests more often may be more likely to ignore potentially genuine test failures.

3.3.3 RQ3: Causes

The middle third of Table 3.2 shows the mean scores of each cause for **SQ5**. The cause with the highest score across all participants was **SQ5.3**. This suggests that improper setup and teardown is the most frequent cause of flaky tests. Flakiness caused by network issues (**SQ5.5**) and unknown reasons (**SQ5.10**) had the second and third highest scores, respectively. This indicates that the causes of many flaky tests go undiagnosed by developers. The lowest scoring was **SQ5.8** concerning floating point issues.

Comparing the participants who experienced flaky tests at least monthly to those who did not, there is agreement that **SQ5.3**, **SQ5.5**, and **SQ5.8** were the first, second, and least most common causes, respectively. The greatest difference in score is for **SQ5.1**. It could be that those participants who said they experience flaky tests on a less than monthly basis do not work on projects that heavily rely on asynchronicity. This could also explain why these particular participants do not experience flaky tests frequently, since the participants who do also scored this cause relatively highly. The differences in mean scores between participants with at least 13 years experience and those with fewer are comparatively small.

According to these results, time and date (**SQ5.7**) appears to be a fairly uncommon cause of flaky tests. Despite this, two participants made strong statements about how time and date logic is a significant cause of flaky tests in **SQ11t6**. P28 said “date handling is the worst thing I have ever had to program around”. This suggests that if a project does rely on time and date logic, this is likely to be a significant cause of the flaky tests of the project’s test suite.

After the thematic analysis for **SQ6**, I identified the following themes in order of prevalence: **SQ6t1: External artifact:** *An issue in an external service, library, or other artifact, that is outside the scope and control of the software under test.* As a potential cause of flaky tests, P8 reported “third-party artifacts, services, or dependencies ... which you do not have full control of ...”. Responses of this prevalent theme were split between highlighting instabilities in remote services (in some instances a special case of **SQ5.5**), and issues in third-party libraries. The common aspect is that participants did not have control over the external artifact. On the external services side, **SQ11t7** is a special case of this theme, further evidencing its prevalence.

SQ6t2: Environmental differences: *Environmental differences between local development machines and remote build machines.* P21 referred to “environmental differences in local vs CI like different Java Virtual Machine (JVM) defaults.” Almost all the responses in this theme made reference to CI. P97 offered a more nuanced explanation, highlighting how file system latency and concurrency-related issues may cause code to behave differently on a CI system. This theme is a special case of **SQ11t4** and directly supports **SQ2t1**.

Table 3.2: Results of the numerical analysis of the responses for **SQ4**, **SQ5**, and **SQ7**. The five “Mean Score (Rank)” columns are the mean scores of each impact, cause, or action for the four specific populations and all participants (All). In each case, the ranks in descending order of mean score are in parentheses. The final column visualises the distribution of responses.

Question	Mean Score (Rank)					All Distribution (All)
	\geq Monthly (SQ3)		\geq 13 Years (SQ10)			
	Yes	No	Yes	No		
RQ2: Impacts						
SQ4.1: Reliability	2.43 (4)	2.47 (2)	2.49 (4)	2.41 (4)	2.45 (4)	
SQ4.2: Efficiency	2.53 (3)	2.38 (4)	2.52 (2)	2.42 (3)	2.47 (3)	
SQ4.3: Productivity	2.58 (2)	2.41 (3)	2.52 (2)	2.49 (2)	2.50 (2)	
SQ4.4: Confidence	2.18 (6)	2.25 (5)	2.27 (5)	2.17 (5)	2.21 (5)	
SQ4.5: CI	2.63 (1)	2.63 (1)	2.68 (1)	2.59 (1)	2.63 (1)	
SQ4.6: Ignore	2.32 (5)	1.96 (7)	2.21 (6)	2.11 (6)	2.16 (6)	
SQ4.7: Reproduce	2.05 (7)	2.14 (6)	2.07 (7)	2.11 (6)	2.09 (7)	
SQ4.8: Differentiate	1.70 (8)	1.85 (8)	1.76 (8)	1.77 (8)	1.76 (8)	
RQ3: Causes						
SQ5.1: Waiting	1.48 (3)	1.08 (5)	1.26 (4)	1.34 (4)	1.30 (4)	
SQ5.2: Concurrency	1.27 (5)	0.95 (7)	1.24 (5)	1.02 (5)	1.12 (5)	
SQ5.3: Setup/teardown	1.73 (1)	1.64 (1)	1.84 (1)	1.56 (1)	1.69 (1)	
SQ5.4: Resources	0.82 (7)	0.97 (6)	0.93 (7)	0.85 (7)	0.89 (7)	
SQ5.5: Network	1.63 (2)	1.21 (2)	1.53 (2)	1.36 (2)	1.44 (2)	
SQ5.6: Random	0.69 (8)	0.70 (9)	0.68 (9)	0.70 (8)	0.69 (9)	
SQ5.7: Time/date	1.01 (6)	1.12 (4)	1.11 (6)	1.02 (5)	1.06 (6)	
SQ5.8: Floating point	0.33 (10)	0.66 (10)	0.59 (10)	0.37 (10)	0.48 (10)	
SQ5.9: Unordered	0.69 (8)	0.78 (8)	0.84 (8)	0.63 (9)	0.73 (8)	
SQ5.10: Unknown	1.45 (4)	1.16 (3)	1.29 (3)	1.35 (3)	1.32 (3)	
RQ4: Actions						
SQ7.1: No action	1.40 (4)	0.91 (5)	1.41 (4)	1.01 (4)	1.19 (4)	
SQ7.2: Re-run	2.81 (1)	2.46 (2)	2.68 (1)	2.66 (1)	2.67 (1)	
SQ7.3: Document	1.58 (3)	1.67 (3)	1.59 (3)	1.65 (3)	1.62 (3)	
SQ7.4: Delete	0.86 (7)	1.06 (4)	1.09 (5)	0.80 (7)	0.94 (5)	
SQ7.5: Quarantine	0.74 (8)	0.79 (7)	0.81 (7)	0.73 (8)	0.77 (8)	
SQ7.6: Mark skip	0.98 (5)	0.85 (6)	1.04 (6)	0.84 (5)	0.93 (6)	
SQ7.7: Mark re-run	0.96 (6)	0.55 (8)	0.74 (8)	0.84 (5)	0.79 (7)	
SQ7.8: Repair	2.23 (2)	2.64 (1)	2.53 (2)	2.31 (2)	2.41 (2)	

SQ6t3: Host system issues: *Problems regarding the machines running the test suites.* In the words of P155, the most common aspect of this theme is “changes in hardware that the code and tests are running on”. In many instances, this is a hardware analogue of **SQ6t2**, where a change in a machine is the cause of the flakiness.

SQ6t4: Test data issues: *Issues originating from the data used by test cases.* Some participants described flakiness caused specifically by test data. Most of these responses were brief and made reference to test data that was “deteriorated”, “changing”, or “external”.

SQ6t5: Resource exhaustion: *Limited computational resources, such as memory and storage.* P49 wrote “unrelated system load on a shared resource causing low-level timeouts”. P133 also made reference to system load from unrelated processes, giving antivirus software as an example. Other responses leaned more towards test cases that consume too many resources themselves. This theme is distinct from **SQ5.4**, which is specifically about mismanagement.

SQ6t6: OS differences: *Differences between operating systems (OS) or different versions of the same OS.* P62 described their experience after upgrading to a later version of Windows — “user interface (UI) changes with new OS. Things like EggPlant, that uses graphics. Moving to a new version of Windows (I think), changed the battleship gray ever so slightly and failed our UI tests”. P76 explained how filesystem differences between OSes can cause flaky tests.

SQ6t7: Virtual machines: *Complications arising from the use of virtual machines or containers.* Put simply by P52, “the automation of virtual machines is asking for trouble”. Vagrant and Docker were specific technologies referred to by the participants.

SQ6t8: UI testing: *Non-determinism inherent to the testing of UIs.* P147 wrote “UI not being in the expected state, i.e., keyboard not closed, or animations not completed when checking results”. P100 specifically described how “random quirks in how Selenium works” caused them to have flaky tests. In many instances, this theme is a special case of **SQ5.1**, since UI test cases often have to wait for a specific element of a UI to be in the correct state [132].

SQ6t9: Conversion issues: *Inconsistencies when converting between data representations.* In the words of P48, “in my code that tests database interactions, I’ve run into issues where my coding language has more time precision than my storage ...”. P103 relayed a similar experience regarding timestamps. While both referred to time, this theme is applicable to any data type.

SQ6t10: Timeouts: *Test execution exceeding a time limit and being prematurely terminated by the test runner.* P76 wrote “not waiting long enough for an environment to be set up”. P79 referred to input and output operations occurring within a specific time limit.

I identified eight themes after analysing the StackOverflow threads. In order of prevalence:

Ct1: UI Timing: *Test case does not wait for a user interface to be in the correct state.* This theme is a subset of **SQ5.1** regarding general asynchronicity and a special case of **SQ6t8** pertaining specifically to timing. This theme is related to **SQ6t3** since the execution speed of the machine is likely to significantly impact any timing issues.

Ct2: Logic error: *Error in the logic of the test code or the code under test.* This theme is broadly characterised by an oversight or misunderstanding on the part of the author of the test case. In one specific instance, a test case used an inappropriate method to wait for a condition in a UI [188]. This led to flakiness by **Ct1**, though since the root cause was that the developer misunderstood the use case of the waiting method, I placed this thread in **Ct2**.

Ct3: Shared state: *Test case depends on state shared with other test cases.* In one thread, the question describes a test case that shares a database connection with other test cases and only passes when executed in isolation [186]. This is an example of an OD test.

Ct4: Unknown: *The cause was never resolved.* Like its counterpart in the survey, **SQ5.10**, this theme was fairly prevalent.

Ct5: Setup/teardown: *Test case does not properly clean up after itself or fails to set up its necessary preconditions.* This is equivalent to **SQ5.3**. While OD tests were the main motivation for **SQ5.3**, the threads in **Ct5** describe test cases that do not appear to be OD. This theme was uncommon yet **SQ5.3** was rated as the most common by participants in the survey. This suggests that the most common causes are not necessarily the hardest for developers to repair.

Ct6: External library: *Bug in a third-party library or package.* This is a special case of **SQ6t1** pertaining specifically to third-party libraries. In one instance, an intermittent

`NullPointerException` from an external package is the cause of flaky tests [182].

Ct7: Resource leak: *Test case does not release acquired resources.* This theme is a subset of **SQ5.4** specifically regarding test cases that do not release resources, rather than general mismanagement. It is similar but not equivalent to **SQ6t5**, since the exhaustion of computational resources is not necessarily due to mismanagement.

Ct8: Improper mocking: *Test case does not mock an object or method correctly.* Like **Ct2**, this theme is unique to the StackOverflow analysis and describes flaky tests caused by improper mocking. A mock is a method or object that simulates the behaviour of its real counterpart to make testing more straightforward [145].

Conclusion for RQ3: Causes: *What causes the flaky tests experienced by developers?* The analysis for **SQ5** suggests that improper setup and teardown is the most common cause of flaky tests. Second to that is network-related issues and third is unknown causes, implying that many flaky tests may go undiagnosed by software developers. Participants rated floating point idiosyncrasies to be the rarest cause. The thematic analysis for **SQ6** revealed additional insights into the causes of flaky tests. The most common theme pertains to issues in external artifacts that the developer has no control over, such as third-party libraries and remote services. Another described differences between local development environments and remote build environments, such as CI. The analysis of the StackOverflow threads, with respect to the causes of flaky tests suggested that timing issues in testing user interfaces were the most common theme. Like the developer survey, the StackOverflow analysis showed that the causes of flaky tests were never resolved in many threads.

3.3.4 RQ4: Actions

The bottom third of Table 3.2 presents the numerical analysis for **SQ7**. The most common action as scored by all participants was to simply re-run the failing build (**SQ7.2**). The second most common was to attempt to repair the flaky test (**SQ7.8**). After these two, there is a significant drop in mean score for the remaining actions. This implies that re-running the build and attempting to repair the flaky test are generally the most common actions developers take when encountering flaky tests. The greatest difference in score between the participants who experienced flaky tests at least monthly and those who did not was for **SQ7.1**. This suggests that developers who experience flaky tests more often are more likely to take no action. There is also a considerable difference for **SQ7.8**, implying that developers who experience flaky tests less frequently are more likely to attempt to repair them. Furthermore, **SQ11t8** indicates that the costs of repair are too great for the potential gains. Arguably, this is less applicable when developers rarely experience flaky tests, which could partially explain the differences in **SQ7.1** and **SQ7.8**.

The thematic analysis for **SQ8** identified the themes regarding actions, in order of prevalence: **SQ8t1: Emotive response:** *An expression of anger or some other emotion.* This theme is generally equivalent to **SQ11t2**, but specifically in the context of a direct response to flaky tests.

SQ8t2: Alert proper person: *Inform other member or members of the development team about the flaky test.* In the words of P52, “tell the person who maintains that codebase”. This theme is similar to **SQ7.3** but is more direct than just documenting the flaky test.

SQ8t3: Reorder tests: *Adjust the order of the test cases.* P7 said “reorder tests to fail faster” and P111 said “reorder tests in case they are order-dependent”. The former seems to be referring to test case prioritisation [131]. The latter is talking about OD tests, but rather than repairing them they are seeking to find a test run order that does not manifest their flakiness [91].

SQ8t4: Repair resource: *Ensure a resource is in the correct state.* summarised by P8, “when the test depends on the global state ... the test needs to be neither deleted/skipped, nor repaired, but rather, the state of the resource needs to be repaired ...”. This theme is arguably a manifestation of **SQ11t5**, since the flaky test highlights a flaw in the software under test.

SQ8t5: Rewrite code under test: *Modify the code under test, as opposed to the test code.* P133 said “rewrite problematic code to make it more testable”. This has clear links with **SQ2t2**, since

it proves that a flaky test can highlight issues in the code under test. It is also a manifestation of **SQ11t5**, for the same reason as **SQ8t4**.

In order of prevalence, the StackOverflow analysis resulted in the following action themes:

At1: Fix logic: *Repair a logic error.* All instances of this theme address an instance of **Ct2**. Given that **Ct2** is generally about API misuse, or inappropriate use of specific elements of an API, answers in **At1** typically highlight the correct API usage or recommend a more appropriate method for a particular purpose [188].

At2: Wait for condition: *Add an explicit wait for a condition.* These answers address most instances of **Ct1** and prescribe waiting for a specific condition, rather than a fixed time delay.

At3: Add mock: *Mock out an object or method.* This theme directly addresses **Ct8** but is also applicable to many other causes, such as **Ct1**. In one instance, the answer recommends mocking to address a timing issue that causes flakiness in a user interface test [192].

At4: Add/adjust external library: *Use a third-party library or adjust the version of a library already in use.* All answers in this theme address an instance of **Ct6**. In one specific thread, the answer highlights the latest version of a particular third-party library that previously contained a bug that was causing the flakiness [182].

At5: Fix setup/teardown: *Repair insufficient setup or teardown procedure.* This directly addresses **Ct5** by suggesting changes to setup and/or teardown methods that were causing flakiness.

At6: Isolate state: *Remove dependency on a shared state.* This theme mostly addresses **Ct4**, but not always. One accepted answer suggests decoupling shared database connections [186].

Conclusion for RQ4: Actions: *What actions do developers take against flaky tests?* For **SQ7**, the analysis revealed that re-running the failing build and attempting to repair the flaky test were the most common actions as rated by the participants. The remaining actions scored significantly lower, indicating that developers are unlikely to perform them. The findings also suggested that developers who experience flaky tests more often are more likely to take no action in response to them. While not a bona fide action, the thematic analysis for **SQ8** showed that an emotive response was very common among the participants. Other themes involved alerting another member of the development team, reordering test cases, or repairing aspects of the software under test, but not the flaky test itself. The thematic analysis of the StackOverflow threads demonstrated that many action themes directly address a single cause theme.

3.4 Recommendations

Table 3.3 lists my six recommendations. I found that **SQ2t1**, the most common theme in **SQ2**, extends my proposed definition of flaky tests to consider factors beyond the code of the test case and the code under test, particularly the execution environment. Furthermore, **SQ6t2** represents environmental differences as a cause of flaky tests and is a special case of **SQ11t4**. This supports a line of research that considers how changes in the implementations of third-party libraries can manifest flaky tests [61, 140, 174]. These results and previous studies are the basis for **R1**. The second most common theme in **SQ2**, **SQ2t2**, represents the idea that flaky tests can indicate that the code under test is flawed. Following the thematic analysis for **SQ8**, I identified **SQ8t4** and **SQ8t5**, regarding repairing a resource and the code under test respectively, as actions in response to flaky tests. Furthermore, **SQ11t5** relays the concept that flaky tests have utility. These results form the foundation of **R2**, along with one of the findings of Eck et al. [37], that, for certain types of flaky test, developers sometimes considered the cause to originate from the code under test.

For **SQ4**, I found that participants strongly agreed that flaky tests hinder CI. This is the motivation for **R3**, along with the findings of Hilton et al. [71], who asked developers to estimate the number of weekly failing CI builds caused by genuine and flaky test failures. They found no significant difference between the two estimates. This also supports Durieux et al. [33], who found that 47% of previously failing CI builds that were manually restarted passed without changes, suggesting the influence of flaky tests. Previous studies identified waiting for asynchronous events (**SQ5.1**) and concurrency (**SQ5.2**) to be the most common causes of flaky tests [37, 105, 136].

Table 3.3: My recommendations. Relevant to researchers (🎓) and developers (</>).

Recommendation	Supported by	
	Results	Literature
</> R1: Consider beyond code. The definition of a flaky test should include factors beyond the test case code or the code under test, such as properties of the execution environment. Developers should consider the behaviour of test cases in different environments, particularly when going from a local environment to CI.	SQ2t1 , SQ6t2 , SQ11t4	[61, 140, 174]
</> R2: Not completely useless. Flaky tests may indicate a flaw in the code under test or another aspect of the software system. Therefore, developers should not write them off as completely useless.	SQ2t2 , SQ8t4 , SQ8t5 , SQ11t5	[37]
🎓 R3: Impact on CI. Flaky tests can become an obstacle to the effective deployment of CI. Researchers should consider the creation and evaluation of new approaches to better mitigate this trend.	SQ4.5	[33, 71]
</> R4: Careful setup/teardown. Insufficient setup and teardown is a common cause of flaky tests. Developers should exercise particular care when writing setup and teardown methods for their test suites.	SQ5.3 , Ct5 , At5	[14]
🎓 R5: Identify root causes. It is difficult to manually determine the root cause of many flaky tests. Researchers should continue to develop automated techniques for this challenging task [88].	SQ5.10 , Ct4	[105]
</> R6: Repair promptly. The results suggest that participants who said they experienced flaky on at least a monthly basis may be more likely to ignore genuine test failures, more likely to take no action in response to flaky tests, and less likely to attempt to repair them. Therefore, developers should to repair flaky tests as soon as possible after identifying them to avoid them accumulating and potentially being ignored.	SQ4.6 , SQ7.1 , SQ7.8 , SQ11t8	[180]

According to the survey, these causes were only the fourth and fifth most common. I found inadequate setup/teardown (**SQ5.3**) to be the most common and also identified this theme in the StackOverflow analysis (**Ct5** and **At5**), forming the basis of **R4**. However, these studies were based on previously repaired flaky tests, whereas in my case, the participants reported the frequency of each cause according to their experience. It could be that the most common causes of flaky tests are not necessarily the ones that developers prioritise for repair. On the other hand, Bell et al. [14] suggested that the difficulty of writing correct setup and teardown could cause OD tests. However, as attested by **Ct5**, this cause may not always result in a flaky test that is OD.

For **SQ5**, I found that participants scored **SQ5.10**, regarding unknown causes, as the third highest overall. Similarly, I found that the cause of the flakiness was never resolved in many of the StackOverflow threads (**Ct4**). This is reflected by Luo et al. [105], who categorised the causes of the repaired flakiness in 201 commits and found “Hard to classify” to be the second most common category. Taken together, these results are the rationale for **R5**. I found that participants who said they experience flaky tests on at least a monthly basis scored **SQ4.6**, regarding ignoring potentially genuine test failures, considerably higher than those who did not. Martin Fowler [180] wrote that flaky tests have an “infectious” quality, and as they proliferate, developers may ignore test failures in general. Furthermore, the results for **SQ7.1** and **SQ7.8** indicate that developers who experience more flaky tests may be more likely to take no action against them and less likely to attempt to repair them. I therefore suggest **R6** in response to these findings.

3.5 Related Work

Luo et al. [105] performed an empirical study that investigated the causes, manifestations, and fixing strategies of flaky tests. As objects of analysis, they used 201 commits that repaired flaky tests in a range of Apache Software Foundation projects. They introduced ten cause categories that have since been used in subsequent research [37, 59, 89, 136]. They found that the top three causes of flaky tests were asynchronicity, concurrency, and test-order dependency (OD tests). Unlike my study, Luo et al. did not base any of their findings on the self-reported experiences of developers. In that respect, their methodology is closer to the StackOverflow analysis.

Eck et al. [37] performed a related study. They asked 21 developers from Mozilla to classify 200 flaky tests that they had previously repaired and also conducted a broader online survey, receiving 121 responses. Using Luo et al.'s ten categories as a starting point, through their Mozilla study, they identified four additional causes, including overly restrictive assertion ranges and platform dependency (broadly similar to **SQ6t2** and **SQ6t6**, respectively). To keep my results as general as possible, I chose not to focus on any particular organisation in any part of the study. Moreover, I included additional objects of analysis beyond developers' testimonies to limit any self-reporting bias [31]. As part of their broader survey, they asked developers to estimate how often they dealt with flaky tests. Their results are very similar to mine for **SQ3**.

As part of a wider study on the uptake of CI, Hilton et al. [71] deployed a survey at Pivotal Software. Among other questions, the survey asked developers to estimate the number of CI builds failing each week due to genuine test failures and due to flaky test failures. Following a Pearson's chi-squared test, they found no statistically significant difference between the genuine and the flaky distributions. My findings confirm that flaky tests are very prevalent (**SQ3**) and that flaky tests are a hindrance to CI (**SQ4.5**). Gruber et al. [56] also deployed a survey about flaky tests, with a specific focus on the support that developers need from tools.

3.6 Conclusion

I deployed an online survey about flaky tests, not restricted to any organisation, and received 170 responses. It focused on understanding how developers define and react to flaky tests and their experiences of the causes and impacts. I also procured a dataset of 38 StackOverflow threads, upon which I performed thematic analysis to identify further causes and repair strategies with the help of my collaborators. From the findings, I offer six actionable recommendations for both researchers and developers. In summary, these are: (1) developers should consider the behaviour of test cases in different environments, particularly when going from a local environment to a continuous integration environment; (2) developers should not write flaky tests off as completely useless; (3) researchers should consider techniques to better mitigate the impact of flaky tests on continuous integration; (4) developers should exercise particular care when writing setup and tear-down methods; (5) researchers should continue working on automated techniques for determining the root causes of flaky tests; and (6) developers should to repair flaky tests as soon as possible after identifying them.

Having furthered the understanding of flaky tests through reviewing published research and consulting professional software developers, I am well positioned to tackle the goal of mitigating them. In Chapter 2, I identified a growing trend of studies applying machine learning classifiers to the detection of flaky tests (see Section 2.5.1). However, none of them consider the detection of order-dependent flaky tests. In this chapter, I found that developers rated improper setup and teardown as the most common cause of flaky tests, while previous studies have identified this as a primary cause of order-dependent flaky tests [14]. Therefore, an evaluation of machine learning-based detection including order-dependent flaky tests is likely to be of value to developers.

Chapter 4

Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests

The contents of this chapter is based on “O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 93–104, 2022”.

4.1 Introduction

Given the problems associated with flaky tests, the research community has developed automated techniques to detect them. Many of these techniques involve a significant number of repeated test executions and some require extensive instrumentation [15, 17, 35, 47, 90, 144, 175], making them prohibitively expensive for practical deployment in large software projects. This motivated researchers to develop detection techniques based on machine learning classifiers that they trained using static features of test cases, such as their length, complexity, and the presence of particular keywords and identifiers [128, 159]. One study found that combining static features with several dynamically-collected characteristics, like execution time and line coverage, resulted in better detection performance at the relatively minimal cost of a single, instrumented test suite run [8].

Previous studies trained and evaluated classifiers using datasets of flaky tests that do not include order-dependent (OD) flaky tests. Yet, Lam et al. found that over 60% of their detected flaky tests were OD [90], suggesting that previous studies may have labelled a large portion of flaky tests as *non-flaky* when training classifiers. This means they have only considered a subset of the problem of flaky test detection. In Chapter 3, developers rated issues regarding setup and teardown as one of the most common causes of flaky tests (see Table 3.2). Since this is generally associated with OD flaky tests [14], it would be beneficial to developers to consider them in evaluations of machine learning classifiers. Given the difficulties caused by OD flaky tests to techniques in software testing research, their efficient detection has significant benefits for researchers too (see Section 2.4). Furthermore, prior studies have only evaluated a limited range of features while the literature has identified many more test case characteristics that may be indicative of flakiness (see Section 2.3). Without further evaluation of a wider range of features, machine learning classifiers cannot be used to their full potential for detecting flaky tests.

This study evaluates the performance of 54 pipelines of data preprocessing, data balancing, and machine learning classifiers for detecting flaky tests in 26 open-source Python projects. Given previous successes with random forest classifiers [8, 128, 143], it focuses on the decision tree classifier and ensemble models thereof [51, 137]. It also introduces FLAKE16, a new feature set

for encoding test cases using seven metrics from a previously established feature set [8] and nine additional metrics, including the depth of the abstract syntax tree of the test case code and the maximum memory usage during test case execution. The results show that FLAKE16 offered a 13% increase in overall F1 score compared to the previous feature set when detecting non-order-dependent (NOD) flaky tests. The experiments also study the same machine learning pipelines with both feature sets for the task of detecting OD flaky tests. In this setup, FLAKE16 offered a 17% increase in the overall F1 score. Finally, the study examines the impact of each FLAKE16 feature on the classifiers’ predictions, revealing that the peak number of concurrently running threads and the number of read- and write-related system calls during test execution are the most valuable features for detecting NOD and OD flaky tests, respectively.

In summary, the main contributions of this study are:

1. **Feature Set (Section 4.2):** A new feature set for machine learning-based flaky test detection, FLAKE16. The evaluation demonstrated an improved detection performance for both NOD and OD flaky tests compared to a previous feature set.
2. **Evaluation (Section 4.3):** My evaluation of 54 machine learning pipelines is the first to consider the detection of OD flaky tests, offering a more complete assessment of the applicability of machine learning to the problem of flaky test detection.
3. **Findings and Implications (Sections 4.4 and 4.5):** Leveraging the empirical results, the study surfaced findings with implications relevant to both the research community and software developers, including the most impactful test case metrics for detecting flaky tests.
4. **Dataset:** To collect the data required to perform the experiments, I developed a comprehensive framework of tools, `Flake16Framework`. To identify flaky tests, I used `Flake16Framework` to execute 5,000 times the test suites for 26 programs containing over 67,000 test cases. Supporting the replication of this study’s results and further investigations into machine learning for flaky test detection, I make `Flake16Framework` and all of the data available in the replication package [218].

4.2 The FLAKE16 Feature Set

Alshammari et al. [8] proposed a range of features for encoding test cases in machine learning-based flaky test detection and split them into two groups. These were eight boolean features indicating the presence of test smells [154], and eight numerical features measuring a mixture of static and dynamic test case properties. They found the test smell features to be of limited value and excluded them from their evaluation of their flaky test detection framework, FLAKEFLAGGER. One of the remaining eight features, the total number of covered production classes, was not applicable in the context of this study. This is because the dataset of test cases used by Alshammari et al. are from Java projects [16] and mine are from Python projects. In Java, classes are a central construct for building programs, whereas in Python, they are less critical and it is possible to write programs without them [197]. I refer to the remaining seven features as the FLAKEFLAGGER feature set, which is subsumed by FLAKE16. One of these features captures the “churn” of the lines covered by a test case, that is, how frequently they are changed. This requires a window of past commits to consider. Alshammari et al. evaluated eight windows and found 75 commits to be the most informative, and thus I selected this value for this study. Beyond these seven features, FLAKE16 contains nine more static and dynamic test case metrics, with Table 4.1 providing a summary.

Several empirical studies identified files as a potential vector for OD flaky tests to arise [15, 17, 47, 105, 175]. In particular, Zhang et al. [175] found that 39% of OD flaky tests were caused by side effects left behind by other test cases in external resources, such as files and databases. Furthermore, flaky tests specifically caused by complications during input and output operations were one of the flaky test categories presented by Luo et al. [105]. This motivated the inclusion of *read count* and *write count* in FLAKE16. These measure the number of read- and write-related

Table 4.1: The metrics of FLAKE16. The **FF** column indicates if the feature is also part of the FLAKE-FLAGGER feature set. The **Static** column indicates if the feature can be measured without executing the test case. The **Impact Rank** columns are the ranks of each feature in descending order of impactfulness for detecting both NOD and OD flaky tests (see Figure 4.2 for more details).

#	Feature	Description	FF	Static	Impact Rank	
					NOD	OD
1	Covered Lines	Number of lines covered.	✓		8	6
2	Covered Changes	Total number of times each covered line has been modified in the last 75 commits.	✓		3	5
3	Source Covered Lines	Number of lines covered that are not part of test cases.	✓		7	7
4	Execution Time	Elapsed wall-clock time of the test case execution.	✓		5	9
5	Read Count	Number of times the filesystem had to perform input [241].			6	2
6	Write Count	Number of times the filesystem had to perform output [241].			4	1
7	Context Switches	Number of voluntary context switches.			10	8
8	Max. Threads	Peak number of concurrently running threads (excluding the main thread).			1	11
9	Max. Memory	Peak memory usage.			11	4
10	AST Depth	Maximum depth of nested program statements in the test case code.		✓	2	13
11	Assertions	Number of assertion statements in the test case code.	✓	✓	14	3
12	External Modules	Number of non-standard modules (i.e., libraries) used by the test case.	✓	✓	16	16
13	Halstead Volume	A measure of the size of an algorithm’s implementation [7, 126, 129].		✓	15	14
14	Cyclomatic Complexity	Number of branches in the test case code [52, 126, 129].		✓	12	12
15	Test Lines of Code	Number of lines in the test case code [126, 129].	✓	✓	9	10
16	Maintainability	A measure of how easy the test case code is to support and modify [164, 279].		✓	13	15

system calls during test execution. Another finding that many empirical studies have in common is that asynchronous operations and concurrency are frequent causes of flaky tests [37, 89, 105, 136]. Thus, I incorporated *context switches* and *maximum threads* into FLAKE16. The former measures the number of *voluntary* context switches performed during test case execution. These occur when a process gives up its CPU time because it has nothing to do, such as when a test case sleeps for a fixed amount of time. Previous studies have identified this as a hallmark of flaky tests in the asynchronous category [37, 105]. I also integrated *maximum memory* into FLAKE16. This feature measures the peak memory usage during test case execution, a property identified by an author of the Google Testing Blog to be correlated with the likelihood of a test case being flaky [243].

The FLAKE16 feature set also contains four additional static metrics that aim to capture the size and complexity of the test case code. A recent study identified this general property to be a possible indicator of flaky tests [129]. With that said, another recent study cast doubt on the reliability of various code complexity metrics for measuring program comprehension difficulty [126]. Nevertheless, this does not necessarily imply that they would be of no use for detecting flaky tests, so this study evaluates them. The first of these is *Abstract Syntax Tree (AST) depth*. Specifically, this feature measures the maximum depth of nested program statements, such as `if` statements and `for` loops. The second is *Halstead volume*, which attempts to capture the “size” of an algorithm’s implementation. Where N is the *total* number of operators and operands in the test case code and η is the number of *distinct* operators and operands, Halstead volume is given

by $N \log_2(\eta)$ [7]. The third static metric is *cyclomatic complexity*, which measures the number of branches in a piece of code [52]. In Python, and many other programming languages, an `if` statement corresponds to a branch and so would increase the cyclomatic complexity by 1. Other examples of branches include `for` and `while` statements, since they both evaluate a condition before every iteration. The fourth metric is *maintainability*. This is an empirical measure of how easy a piece of code is to support and modify [164]. There are several formulations, though I used the one implemented by the `Radon` library [279]. The library calculates maintainability (MI) in terms of Halstead volume (V), cyclomatic complexity (G), test lines of code (L), and the percentage of lines that are comments, converted to radians (C):

$$MI = \max \left[0, 100 \frac{171 - 5.2 \ln V - 0.23 G - 16.2 \ln L + 50 \sin \sqrt{2.4 C}}{171} \right] \quad (4.1)$$

4.3 Evaluation

I designed and conducted experiments to answer the following three research questions:

RQ1. Compared to the features used by `FLAKEFLAGGER`, does the `FLAKE16` feature set improve the performance of flaky test case detection with machine learning classifiers?

RQ2. Can machine learning classifiers effectively detect order-dependent flaky test cases?

RQ3. Which features of `FLAKE16` are the most impactful?

4.3.1 Data Collection

To evaluate the performance of any machine learning classifier for detecting flaky tests, I needed a labelled dataset of test cases. To that end, I sampled 26 popular Python projects, most of which are considered critical to open-source infrastructure [260]. In total, these 26 projects, listed in Table 4.2, gave me a dataset of over 67,000 test cases. In order to train and evaluate a machine learning classifier, I needed to label each test case as *non-flaky*, *NOD flaky*, or *OD flaky*. To that end, I created a framework of tools, called `Flake16Framework`, to automatically execute each project’s test suite 2,500 times in a consistent order and an additional 2,500 times in a random order. For reproducibility and isolation between test suite runs, `Flake16Framework` installs each project inside of a fully-specified virtual environment [195] to produce a Docker image [215], which it uses to create a separate container for each test suite run. The framework also records the outcome (i.e., pass or fail) of every test during each test suite execution. It labels a test as *NOD flaky* if it has an inconsistent outcome during the runs in a consistent order. Otherwise, it labels a test as *OD flaky* if it has an inconsistent outcome during the random order runs. Failing that, it labels a test as *non-flaky*. This is an established practice for identifying flaky tests [59, 92].

Given the non-deterministic nature of flaky tests, it is impossible to label a test case as non-flaky with complete certainty. Naturally, confidence increases with the number of test suite runs, but so too does the computational cost. Alshammari et al. [8] executed test suites 10,000 times. Based on their findings, the cumulative number of detected flaky tests appears sublinearly related to the number of test suite runs. In other words, continuing to re-execute a test suite gives diminishing returns with respect to the confidence of labelling a test case as non-flaky. This study confirms these findings, for both NOD and OD flaky tests, as illustrated by the curves in Figure 4.1. As such, I selected a smaller number of test suite runs to reduce the time to finish the labelling process. Despite this, labelling still took over four weeks of computational time on a computer with a 24-core AMD Ryzen 5900X CPU.

As well as having labels for each test case, I also needed to measure values for each of the metrics of `FLAKE16`. To that end, I designed `Flake16Framework` to perform the necessary static analysis on the source code of every test case and to instrument test case execution to collect the dynamic features. I implemented this with the help of several existing Python libraries. To collect most of the static metrics, I used the `Radon` library [279]. To determine the number of external modules used by a test case, I implemented my own approach that analyses the AST of a test case. To measure line coverage data, I used `Coverage.py` [211]. For the majority of the remaining

Table 4.2: The 26 open-source Python subjects. The **Stars** column is the number of times a GitHub user has indicated their interest in the project [270]. The **Tests** column is the total number of test cases. The **NOD** and **OD** columns are the number of non-order-dependent and order-dependent flaky tests.

GitHub Repository	# Stars	# Tests	# NOD	# OD
apache/airflow	23175	3458	66	293
celery/celery	17952	2365	-	15
conan-io/conan	5274	3707	-	13
encode/django-rest-framework	21906	1402	-	1
spesmilo/electrum	5154	544	1	1
Flexget/Flexget	1342	1335	1	4
fonttools/fonttools	2850	3456	1	42
graphql-python/graphql	6810	347	-	1
facebookresearch/hydra	4861	1540	-	19
HypothesisWorks/hypothesis	5379	4386	5	6
ipython/ipython	14982	846	6	304
celery/kombu	2221	1025	2	23
apache/libcloud	1788	9840	3	133
Delgan/loguru	9838	1255	4	21
mitmproxy/mitmproxy	24702	1231	-	17
python-pillow/Pillow	8983	2583	-	26
PrefectHQ/prefect	6897	7038	25	20
PyGithub/PyGithub	4664	711	-	4
Pylons/pyramid	3593	2633	-	4
psf/requests	46050	537	5	-
scikit-image/scikit-image	4525	6281	-	12
mwaskom/seaborn	8772	1028	1	8
pypa/setuptools	1439	704	1	23
sunpy/sunpy	629	2072	-	2
urllib3/urllib3	2788	1900	15	1
xonsh/xonsh	5133	4782	9	19
Total	241707	67006	145	1012

dynamic features, I used `psutil` [264]. In keeping with previous work [8], `Flake16Framework` executed each of the 26 test suites just once to measure these values to keep its computational cost as low as possible. While there may be some expected variance in these values, I leave it as future work to investigate if the repeated measurement of these features improves the performance of flaky test detection (see Section 5.6.2).

4.3.2 Data Preprocessing

Preprocessing of raw feature data is a typical component of machine learning pipelines [54, 171]. To that end, I evaluated two common data preprocessing techniques. The first was *scaling* (also known as *standardisation*), which, for each feature, involves subtracting the mean over the entire dataset and dividing by the standard deviation. This has the effect of “centering” the distribution of each feature with a mean of zero and a variance of one, such as a standard normal distribution. This is a common requirement for many machine learning models [196]. The second was *principal component analysis* (PCA) [1]. This is a technique used to transform a dataset such that each new feature corresponds to a *principal component*. The principal components of a dataset can be thought of as an ordered set of orthogonal vectors representing axes that best capture the variance of the data. The first principal component captures the most variance and the subsequent components capture increasingly less. A common use of PCA is to reduce the number of features in a dataset while sacrificing as little data as possible. This is known as *dimensionality reduction* [156]. Because the principal components are orthogonal to one another, PCA also decorrelates the

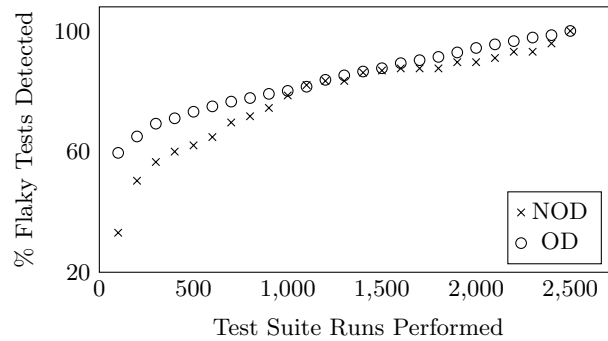


Figure 4.1: The relationship between the number of test suites runs performed by `Flake16Framework` and the percentage of flaky tests it identified, both NOD and OD. The relationship in both cases is sublinear.

features. Since the dimensionality of the dataset is relatively low, and Table 4.3 shows that many of FLAKE16’s features are correlated, decorrelation is my primary use case for PCA.

4.3.3 Data Balancing

As shown by Table 4.2, the number of non-flaky tests in the dataset vastly outnumbers both the NOD and OD flaky tests. Training machine learning classifiers with imbalanced data potentially limits their performance [102, 148]. To address this, I evaluated five data balancing techniques. Data balancing techniques can be split into two categories: those that *undersample* (reduce) the majority class (non-flaky), and those that *oversample* (increase) the minority class (flaky). I evaluated two undersampling, one oversampling, and two combined techniques. The first undersampling technique removes the non-flaky samples within *Tomek links* of the dataset. A Tomek link occurs between two samples when a sample of one class is the nearest neighbour of a sample of the other [153]. The second technique, *edited nearest-neighbors*, removes non-flaky samples whose nearest neighbors are all flaky. With a neighbourhood of only one sample, edited nearest-neighbors is equivalent to the previous technique. I used a neighbourhood of three, the implementation’s default. For oversampling, I evaluated the *synthetic minority oversampling technique* (SMOTE) [23], which generates synthetic flaky samples by interpolation. The two combined techniques I evaluated were the combination of SMOTE with Tomek links and edited nearest-neighbors. In both instances, SMOTE is applied first and the undersampling technique acts as a data cleaning method, rather than to undersample the non-flaky samples [11]. To ensure the correctness of these balancing techniques, I used those in the `imbalanced-learn` Python library [240].

4.3.4 Machine Learning Classifiers

For the classification of test cases as flaky or non-flaky, I evaluated three machine learning classifiers. Previous studies have found the *Random Forest* classifier [18] to be particularly performant in this domain [8, 128]. Random forest is an *ensemble* model, which combines many *base models*, in this case *Decision Tree* [137]. Decision tree is a non-parametric model that learns simple *if-then-else* decision rules from the training data, forming a binary tree. In this context, their output is the estimated probability of a test case being non-flaky. The Random Forest classifier trains each Decision Tree on a *random sample with replacement* of the training data, that is, a sample where individual data points can appear more than once. The overall classification is based on the average of their estimated probabilities. A related classifier, *extremely randomised trees* [51], also known as *Extra Trees*, trains trees on random samples *without* replacement and introduces additional randomisation in how they are trained. I evaluated Random Forest and Extra Trees, as well as the base Decision Tree classifier, using the implementations provided by `scikit-learn` [271].

Table 4.3: Spearman rank-order correlation coefficients between each pair of features in FLAKE16. Values range between -1 and 1, with 0 indicating no correlation and -1 or 1 indicating an exact monotonic relationship. A negative value indicates that as the feature in the row increases, the feature in the column decreases. Darker shaded cells indicate a greater magnitude of correlation.

	Covered Lines	Covered Changes	Source Covered Lines	Execution Time	Read Count	Write Count	Context Switches	Max. Threads	Max. Memory	AST Depth	Assertions	External Modules	Halstead Volume	Cyclomatic Complexity	Test Lines of Code	Maintainability
Cov. Lines	1.00	0.72	1.00	0.48	-0.05	0.32	0.50	0.15	0.21	-0.09	-0.07	-0.30	-0.11	-0.08	0.15	0.09
Cov. Changes	0.72	1.00	0.72	0.41	-0.01	0.25	0.35	0.13	0.21	0.04	0.14	-0.14	0.08	0.13	0.17	-0.09
Src. Cov. Lns.	1.00	0.72	1.00	0.48	-0.05	0.32	0.49	0.14	0.21	-0.10	-0.06	-0.31	-0.10	-0.07	0.13	0.08
Exec. Time	0.48	0.41	0.48	1.00	0.55	0.48	0.48	0.30	0.76	0.03	0.17	-0.01	0.13	0.14	0.09	-0.12
Read Count	-0.05	-0.01	-0.05	0.55	1.00	0.40	0.12	0.18	0.73	0.17	0.21	0.30	0.21	0.22	0.12	-0.18
Write Count	0.32	0.25	0.32	0.48	0.40	1.00	0.52	0.32	0.45	0.00	-0.07	-0.06	-0.04	-0.07	0.13	0.04
Con. Switches	0.50	0.35	0.49	0.48	0.12	0.52	1.00	0.31	0.36	-0.11	-0.04	-0.18	-0.05	-0.06	0.09	0.07
Max. Threads	0.15	0.13	0.14	0.30	0.18	0.32	0.31	1.00	0.26	0.05	-0.01	0.05	0.01	0.00	0.10	-0.01
Max. Memory	0.21	0.21	0.21	0.76	0.73	0.45	0.36	0.26	1.00	0.10	0.21	0.16	0.20	0.20	0.06	-0.16
AST Depth	-0.09	0.04	-0.10	0.03	0.17	0.00	-0.11	0.05	0.10	1.00	0.21	0.16	0.24	0.42	0.42	-0.26
Assertions	-0.07	0.14	-0.06	0.17	0.21	-0.07	-0.04	-0.01	0.21	0.21	1.00	0.15	0.73	0.89	0.30	-0.69
Ext. Modules	-0.30	-0.14	-0.31	-0.01	0.30	-0.06	-0.18	0.05	0.16	0.16	0.15	1.00	0.28	0.18	0.18	-0.24
Halstead Vol.	-0.11	0.08	-0.10	0.13	0.21	-0.04	-0.05	0.01	0.20	0.24	0.73	0.28	1.00	0.75	0.35	-0.91
Cyclo. Comp.	-0.08	0.13	-0.07	0.14	0.22	-0.07	-0.06	0.00	0.20	0.42	0.89	0.18	0.75	1.00	0.43	-0.72
Test LoC	0.15	0.17	0.13	0.09	0.12	0.13	0.09	0.10	0.06	0.42	0.30	0.18	0.35	0.43	1.00	-0.39
Maintainability	0.09	-0.09	0.08	-0.12	-0.18	0.04	0.07	-0.01	-0.16	-0.26	-0.69	-0.24	-0.91	-0.72	-0.39	1.00

4.3.5 Methodology

I used the `Flake16Framework` to evaluate all combinations of data preprocessing, data balancing, and machine learning classifier, including no preprocessing and no balancing. This resulted in 54 machine learning pipelines. The framework evaluated these using both the FLAKE16 and the FLAKEFLAGGER feature sets and applied them to two binary classification problems, *non-flaky or NOD flaky* and *non-flaky or OD flaky*, culminating in 216 classifiers. It used *stratified 10-fold cross validation* for classifier training and testing, as done by Alshammari et al. in their evaluation of their FLAKEFLAGGER framework [8]. This creates 10 *folds*, in which 90% of the dataset is used for training and 10% is used for testing. The class balance of each fold roughly follows that of the entire dataset, and `Flake16Framework` applied data balancing to the training set only. This is so the classifier testing accurately reflects the imbalanced nature of the classification problem. After training the classifier, the framework applied it to each test case of the testing set, resulting in a prediction of flaky or non-flaky. Since the testing portion of each fold is unique, after 10 folds every test case in the dataset has a predicted label.

Where flaky is the *positive* label and a predicted label for a test case is *true* if it matches the label assigned to it during data collection, `Flake16Framework` enumerated the number of *false positives*, *false negatives*, and *true positives*, for the test cases of each project and for the entire dataset. From these, it calculated the *precision*, the ratio of true positives to all positives, and the *recall*, the ratio of true positives to the sum of true positives and false negatives. In other words, precision is the fraction of genuine flaky tests among all test cases the classifier labelled as flaky and recall is the fraction of genuine flaky tests that the classifier labelled as flaky. The framework also calculated the *F1 score*, or the harmonic mean of these two metrics. These are all common

metrics used in previous studies of machine learning to detect flaky tests [8, 128, 159]. The F1 score metric is particularly well-suited to imbalanced binary classification problems, like the one in this study, since it penalises significant differences between a classifier’s precision and recall. This is important because a classifier that trivially labels all test cases as non-flaky would achieve maximum recall but very poor precision. I answered **RQ1** by comparing the performance of the best pipeline using the FLAKEFLAGGER feature set to the best using the FLAKE16 set, for both the NOD and OD classification problems. I answered **RQ2** by focusing on the OD problem.

To rank each feature of FLAKE16 in terms of its impact, I used the *Shapely Additive Explanations* (SHAP) technique [103]. This automated technique leverages concepts from game theory to quantify the contribution that a feature has on the output of a machine learning model. SHAP requires a dataset (i.e., a matrix where each row is a data point and each column represents a feature) and a trained model, and it returns a matrix of SHAP values in the same shape as the dataset. Each column of the SHAP values matrix corresponds to the impact that the respective feature had on the decision of the trained model for each data point. In my case, `Flake16Framework` applied SHAP to estimate the contribution of each feature of FLAKE16 to a classifier’s estimated probability of each test case being non-flaky. It did this for the best non-PCA pipelines for both test case classification problems when using FLAKE16.

Since the features of a PCA-transformed dataset correspond to a linear combination of the original features [1], it would be difficult to relate their impact back to the features of FLAKE16. To answer **RQ3**, `Flake16Framework` quantified the impact of each feature for detecting both NOD and OD flaky tests by taking the mean absolute value of each column of the SHAP values matrices for both pipelines. A common alternative technique I could have used is to calculate the *permutation importance* of each feature. Given a trained model, this would involve shuffling the values of each feature across the dataset and measuring the impact this has on the performance of the model when applied to the shuffled dataset (e.g., the F1 score). Yet, this technique can give misleading results when features are correlated [73], as Table 4.3 shows is the case for the features used to predict whether or not a test is flaky.

4.3.6 Threats to Validity

This section considers the potential threats to the validity of this evaluation and discusses how I mitigated them. First, during data collection `Flake16Framework` could have labelled some flaky tests as non-flaky. Given the non-deterministic nature of flaky tests, it is impossible to fully rectify this issue, although I mitigated it by performing as many test suite runs as possible within the limits of the computational resources available. Furthermore, some specific categories of flaky tests are unlikely to be manifested by rerunning alone [105, 140]. The only category I made special arrangements to detect were OD flaky tests; I consider other categories requiring additional means to identify out of the scope of this study. Second, `Flake16Framework` could have contained bugs, which may have impacted the results of the evaluation. To that end, I used well-established Python libraries for the bulk of its functionality. These included `Coverage.py` [211] to measure line coverage, `psutil` [264] to measure many other dynamic properties of test cases, and `scikit-learn` [271] for the classifier implementations. These are all popular open-source projects with many contributors, giving me confidence that any bugs would be identified, documented, and patched in a timely manner. I also wrote unit tests for greater confidence in the correctness of the bespoke elements of `Flake16Framework`. Third, individual projects with significantly more test cases than others could bias the overall results. For example, `airflow` had the highest number of NOD flaky tests at 66, or 264% more than that of the second highest. To resolve this concern, I calculated performance metrics with respect to each individual project.

Table 4.4: Top 10 pipelines for detecting flaky tests. Train time (**Tr.**) and test time (**Te.**) are in seconds.

FLAKEFLAGGER						FLAKE16						
Pre.	Balancing	Classifier	Time			F1	Pre.	Balancing	Classifier	Time		
			Tr.	Te.	F1					Tr.	Te.	F1
NOD												
None	Tom. Links	Ext. Trees	8	0.27	0.46	PCA	SMOTE	Ext. Trees	133	0.57	0.52	
Scaling	None	Ext. Trees	11	0.37	0.44	PCA	SMOTE Tom.	Ext. Trees	100	0.26	0.52	
None	SMOTE Tom.	Ext. Trees	41	0.26	0.43	Scaling	SMOTE Tom.	Ext. Trees	123	0.59	0.51	
None	SMOTE	Ext. Trees	21	0.21	0.43	Scaling	SMOTE	Ext. Trees	139	0.49	0.51	
None	ENN	Ext. Trees	6	0.13	0.43	None	SMOTE	Rand. For.	269	0.22	0.48	
None	None	Ext. Trees	14	0.49	0.43	None	ENN	Ext. Trees	34	0.25	0.48	
Scaling	ENN	Ext. Trees	10	0.46	0.43	PCA	SMOTE ENN	Ext. Trees	146	0.53	0.48	
Scaling	Tom. Links	Ext. Trees	4	0.16	0.43	Scaling	SMOTE Tom.	Rand. For.	209	0.25	0.48	
Scaling	SMOTE Tom.	Ext. Trees	41	0.56	0.42	None	SMOTE Tom.	Ext. Trees	53	0.20	0.48	
PCA	Tom. Links	Ext. Trees	13	0.27	0.42	PCA	SMOTE Tom.	Rand. For.	334	0.32	0.48	
OD												
None	SMOTE Tom.	Ex. Trees	70	1.23	0.47	Scaling	SMOTE	Rand. For.	148	0.52	0.55	
None	SMOTE	Ex. Trees	75	1.16	0.46	Scaling	SMOTE Tom.	Rand. For.	173	0.65	0.55	
None	SMOTE Tom.	Rand. For.	83	0.81	0.46	Scaling	SMOTE	Ex. Trees	150	0.92	0.54	
None	SMOTE	Rand. For.	85	0.28	0.45	Scaling	SMOTE Tom.	Ex. Trees	155	1.49	0.54	
None	ENN	Ex. Trees	13	0.80	0.45	None	SMOTE	Ex. Trees	128	1.24	0.53	
Scaling	ENN	Ex. Trees	13	0.92	0.45	Scaling	ENN	Ex. Trees	37	0.79	0.52	
Scaling	Tom. Links	Ex. Trees	18	1.08	0.45	None	SMOTE Tom.	Ex. Trees	38	0.43	0.52	
Scaling	None	Ex. Trees	21	1.17	0.44	None	Tom. Links	Ex. Trees	27	0.62	0.51	
None	None	Ex. Trees	12	0.69	0.44	Scaling	Tom. Links	Ex. Trees	29	0.79	0.51	
None	Tom. Links	Ex. Trees	14	0.70	0.44	None	ENN	Ex. Trees	27	0.80	0.50	

4.4 Empirical Results

4.4.1 RQ1. Compared to the features used by FLAKEFLAGGER, does the FLAKE16 feature set improve the performance of flaky test case detection with machine learning classifiers?

The top half of Table 4.4 shows the top 10 pipelines for detecting NOD flaky tests with both the FLAKEFLAGGER and FLAKE16 feature sets. The best pipeline with the FLAKE16 feature set was preprocessing with PCA, balancing with SMOTE, and Extra Trees as the classifier. Its F1 score was 13% higher than the best pipeline with the FLAKEFLAGGER feature set. Table 4.5 shows the per-project performance of the best pipelines with both feature sets. This table excludes projects for which I could not calculate precision, recall, or F1 score due to a division by zero. For all the included projects, with the exception of `xonsh`, the F1 score is either unchanged or higher with FLAKE16. Overall, the best pipeline with FLAKE16 had a better trade-off between precision and recall, whereas the best pipeline with the FLAKEFLAGGER feature set had significantly greater precision than recall, which was relatively poor. This result suggests that the FLAKEFLAGGER pipeline was particularly conservative with regard to labelling a test case as flaky. The bottom half of Table 4.4 gives the best pipelines for the OD classification problem. In this case, the best pipeline with FLAKE16 had an F1 score that was 17% greater than the best pipeline with the FLAKEFLAGGER feature set. Table 4.6 gives the per-project scores for these two pipelines. Once again, I excluded projects if I could not calculate precision, recall, or F1 score. For 11 of the 18 projects listed, the F1 score was greater when using FLAKE16. Overall, the best pipeline with FLAKE16 had a recall that was 36% greater than that of the FLAKEFLAGGER feature set and a precision that was unchanged, indicating a clear improvement in the performance.

Table 4.5: For NOD flaky tests, the per-project comparison of the best pipeline for both feature sets. **FP**, **FN**, and **TP** are false positives, false negatives, and true positives. **Pr.** is precision and **Re.** is recall.

Project	FLAKEFLAGGER						FLAKE16					
	FP	FN	TP	Pr.	Re.	F1	FP	FN	TP	Pr.	Re.	F1
airflow	7	37	29	0.81	0.44	0.57	20	30	36	0.64	0.55	0.59
ipython	0	4	2	1.00	0.33	0.50	3	2	4	0.57	0.67	0.62
loguru	1	2	2	0.67	0.50	0.57	1	2	2	0.67	0.50	0.57
prefect	2	17	8	0.80	0.32	0.46	5	12	13	0.72	0.52	0.60
requests	1	3	2	0.67	0.40	0.50	1	3	2	0.67	0.40	0.50
xonsh	4	4	5	0.56	0.56	0.56	3	5	4	0.57	0.44	0.50
Total	16	97	48	0.75	0.33	0.46	49	76	69	0.58	0.48	0.52

Conclusion for RQ1 The FLAKE16 feature set offered a 13% increase in overall F1 score when detecting NOD flaky tests and a 17% increase when detecting OD flaky tests. These results indicate that the FLAKE16 feature set improves machine learning-based flaky test detection performance compared to the FLAKEFLAGGER feature set.

4.4.2 RQ2. Can machine learning classifiers effectively detect order-dependent flaky test cases?

As shown in the bottom half of Table 4.4, the most performant pipeline for detecting OD flaky tests used the FLAKE16 feature set with scaling for preprocessing, SMOTE for balancing, and Random Forest as the classifier, and achieved an F1 score of 0.55. Compared to the best NOD pipeline, its overall precision was 14% lower and its recall was 25% higher, resulting in an F1 score that was just 6% higher. These differences are too marginal to conclude that machine learning classifiers are any better at detecting OD flaky tests than NOD flaky tests, but suggests that their performance at both classification problems was roughly the same. For the best OD pipeline, the per-project F1 scores showed a significant degree of variance, achieving an F1 score of just 0.31 for `conan` and up to 0.79 for `fonttools`. Comparatively, the best NOD pipeline had a much lower per-project variance, though the sample size of projects in this case is rather small to draw any reliable conclusions. This per-project variance is not unique to this study [8, 159], however this is the first to report it in the context of using machine learning to detect OD flaky tests.

Conclusion for RQ2 The performance of the best OD pipeline was broadly similar to that of the best NOD pipeline, suggesting that machine learning classifiers are just as applicable to the task of detecting OD flaky tests as they are to detecting NOD flaky tests.

4.4.3 RQ3. Which features of FLAKE16 are the most impactful?

Figure 4.2 shows each feature of FLAKE16 in descending order of their mean absolute SHAP value in the context of detecting both NOD and OD flaky tests. The lines connecting the boxes indicate how their ranks differ between the two classification problems. As indicated by the number of lines with steep gradients, the difference is significant. For detecting NOD flaky tests, the maximum threads feature was the most impactful by a considerable margin. For OD flaky tests, the number of read- and write-related system calls were the most impactful metrics. All these features are exclusive to FLAKE16, which could partially explain why it improved detection performance compared to the FLAKEFLAGGER feature set. In general, the dynamic features occupy the higher ranks and the static features occupy the lower ranks for both classification problems. This shows that the static features had less influence on the classifiers' decisions, implying that they may be less useful for detecting flaky tests. Clear exceptions to this are the AST depth feature, which was the second most impactful for NOD flaky tests, the number of assertions,

Table 4.6: For OD flaky tests, the per-project comparison of the best pipeline for both feature sets. **FP**, **FN**, and **TP** are false positives, false negatives, and true positives. **Pr.** is precision and **Re.** is recall.

Project	FLAKEFLAGGER						FLAKE16					
	FP	FN	TP	Pr.	Re.	F1	FP	FN	TP	Pr.	Re.	F1
airflow	89	94	199	0.69	0.68	0.69	133	55	238	0.64	0.81	0.72
celery	6	7	8	0.57	0.53	0.55	4	9	6	0.60	0.40	0.48
conan	23	7	6	0.21	0.46	0.29	14	8	5	0.26	0.38	0.31
Flexget	2	3	1	0.33	0.25	0.29	0	3	1	1.00	0.25	0.40
fonttools	19	5	37	0.66	0.88	0.76	21	1	41	0.66	0.98	0.79
hydra	19	9	10	0.34	0.53	0.42	4	13	6	0.60	0.32	0.41
ipython	96	248	56	0.37	0.18	0.25	274	126	178	0.39	0.59	0.47
kombu	6	13	10	0.62	0.43	0.51	1	15	8	0.89	0.35	0.50
libcloud	62	91	42	0.40	0.32	0.35	103	78	55	0.35	0.41	0.38
loguru	4	2	19	0.83	0.90	0.86	6	6	15	0.71	0.71	0.71
mitmproxy	10	11	6	0.38	0.35	0.36	5	12	5	0.50	0.29	0.37
Pillow	14	20	6	0.30	0.23	0.26	21	11	15	0.42	0.58	0.48
prefect	9	16	4	0.31	0.20	0.24	1	16	4	0.80	0.20	0.32
PyGithub	0	1	3	1.00	0.75	0.86	1	1	3	0.75	0.75	0.75
scikit0image	31	3	9	0.23	0.75	0.35	2	7	5	0.71	0.42	0.53
seaborn	11	6	2	0.15	0.25	0.19	1	6	2	0.67	0.25	0.36
setuptools	5	5	18	0.78	0.78	0.78	4	7	16	0.80	0.70	0.74
xonsh	10	12	7	0.41	0.37	0.39	8	13	6	0.43	0.32	0.36
Total	440	569	443	0.50	0.44	0.47	608	401	611	0.50	0.60	0.55

which was third for OD flaky tests, and test lines of code, which occupied the lower-middle ranks in both instances.

Conclusion for RQ3 The most impactful feature when detecting NOD flaky tests was the peak number of concurrently running threads during test case execution. When detecting OD flaky tests, the number of read- and write-related system calls were the most impactful. In general, the dynamic features were more impactful than the static features, though there were notable exceptions to this.

4.5 Discussion

4.5.1 General Classifier Performance

The best pipeline for detecting NOD flaky tests achieved an F1 score of 0.52 and the best for OD flaky tests achieved an F1 score of 0.55. For a binary classification problem with balanced classes, an F1 score of 0.50 can be trivially attained by randomly guessing labels with uniform class probabilities. Yet, both of this study’s classification problems are significantly imbalanced. For the NOD problem, flaky tests account for just 0.02% of the entire dataset. For the OD problem, flaky tests represent 1.5%. In both cases, one would expect uniform random guessing to yield an F1 score significantly lower than 0.5. Considering the NOD problem, one would expect random guessing to render half of the 66,861 non-flaky test cases as true negatives and half as false positives. Similarly, one would expect half of the 145 flaky test cases to become false negatives and the other half true positives. Applying the calculations for precision and recall described in Section 4.3.5, this strategy would score 0.5 and 0.002 respectively, giving an ultimate F1 score of 0.004. The F1 score would be similarly low for the OD problem and for both problems using other trivial approaches, such as guessing according to class prior probabilities or labelling all test cases as non-flaky. Therefore, the two pipelines that use FLAKE16 are significantly better suited to flaky test detection than these trivial approaches.

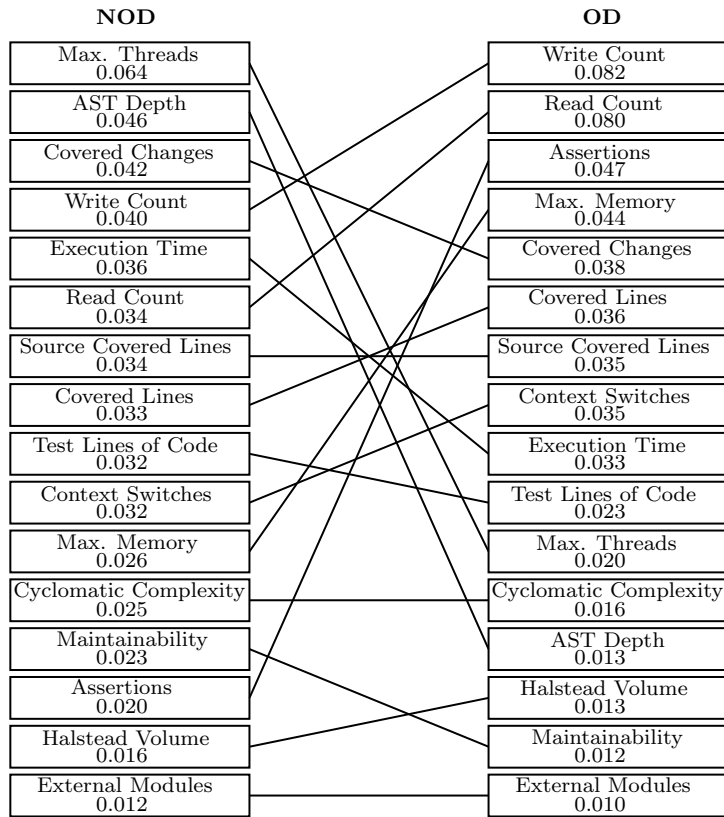


Figure 4.2: Impact of each feature of FLAKE16 for both classification problems. Each box represents a feature and contains its mean absolute SHAP value, of which they are in descending order. The left column contains the values for detecting NOD flaky tests and the right for of detecting OD flaky tests. Features are connected between each column with a line, representing how significantly the ranks differ.

Alshammari et al. [8], who presented the FLAKEFLAGGER framework, recorded an F1 score of 0.66 when detecting NOD flaky tests. This is considerably higher than the F1 score achieved by the best pipeline with the FLAKEFLAGGER feature set in this study, which was 0.46. Yet, Alshammari et al. used an entirely different dataset of tests from projects in the Java programming language, which makes the comparison largely invalid. With this Python dataset, I demonstrated that FLAKE16 improved NOD flaky test detection performance. There is no evidence to suggest that this would not also be the case on Alshammari et al.'s dataset.

4.5.2 Reliability of Performance Metrics

Extrapolating the curves of Figure 4.1 suggests that I would identify more flaky tests of both categories if I had Flake16Framework perform more test suite runs. Since more runs can only result in test case labels transitioning from non-flaky to flaky, true negatives may become false negatives and false positives may become true positives. The effect that this would have on precision, recall, and F1 score would entirely depend on the frequency of both types of change. However, I do not consider it likely that more test suite runs would change the overall conclusion of **RQ1** for two reasons. First, any label transitions would affect the results for the FLAKEFLAGGER feature set the same as they would for FLAKE16. Second, given the sublinear curves, I would expect to identify increasingly fewer flaky tests as Flake16Framework performed more runs, meaning that any changes in the F1 scores would be increasingly small.

4.5.3 Impact of Features

The results for **RQ3** indicate that maximum threads was the most impactful feature when detecting NOD flaky tests. This is unsurprising given the prevalence of flaky tests caused by asynchronicity and concurrency, as reported in the literature [37, 89, 105, 136]. Naturally, both of these causes imply multiple running threads during test case execution. For OD flaky tests, the results indicate that the number of read- and write-related system calls were the most impactful. Similarly, this could be explained by the relationship between filesystem activity and OD flaky tests that has been described in previous studies [15, 17, 47, 105, 175]. Interestingly, these two features were also highly impactful when detecting NOD flaky tests. A possible explanation for this is that input and output operations may be performed asynchronously [210], which has been established as a common cause of NOD flaky tests [105].

Alshammari et al.'s evaluation suggested that execution time was the most informative feature that they considered for their FLAKEFLAGGER framework [8]. In the results of this study, execution time was the fifth most impactful feature when detecting NOD flaky tests, but was considerably less impactful for OD flaky tests. They also determined that the three coverage features were highly informative. Similarly, I found these to occupy the upper-middle ranks for both classification problems, supporting the notion that they are valuable features for detecting flaky tests. In general, one would expect features such as execution time, the three coverage features, and maximum memory to be higher for larger and more complex test cases. I hypothesise that the impactfulness of this group of features is simply a consequence of the intuition that the more a test case is doing, the greater the margin for both error and flakiness.

Many of the static features of FLAKE16, such as cyclomatic complexity, Halstead volume, and maintainability, appeared to have the lowest impact for both classification problems. A recent study cast doubt on the fitness for purpose of Halstead volume and other code complexity metrics [126]. This could be the reason why they appeared to be of limited value in the context of flaky test detection. This is despite another recent study finding that the Halstead volume of flaky tests was greater than non-flaky tests to a statically significant degree, albeit with a small effect size [129]. With this result in mind, concluding that static features are generally less valuable than dynamic features could be misguided, especially since I identified three static features that appeared to make a considerable contribution to the classifiers' predictions.

4.5.4 Impact of Preprocessing, Balancing, and Classifier Choice

Table 4.4 does not indicate any clear best choice of preprocessing or balancing. For both the FLAKEFLAGGER and FLAKE16 feature sets, pipelines with different preprocessing and balancing are in the top 10 for both classification problems with only small differences in F1 score. Most interestingly, given the significant class imbalances, pipelines with no data balancing are in the top 10 for both classification problems with the FLAKEFLAGGER features. In general, these results indicate that Extra Trees was the better choice of classifier, though this is not definitive since Random Forest was best for detecting OD flaky tests with FLAKE16, though in comparison to extra trees the difference in F1 score is minimal. Overall, the only reliable conclusions I can draw from these findings is that data balancing mostly improves detection performance, though there is no clear best technique, and Extra Trees appears to have a slight edge over random forest. When compared to Random Forest, Extra Trees trades increased *bias* for reduced *variance* [51]. Having increased bias means the model may fail to recognise relationships between feature data and labels, known as *underfitting*. Having reduced variance means the model may be less sensitive to noise and outliers, avoiding *overfitting*. It could be that the particular bias-variance trade-off of Extra Trees makes it generally more suited to the specific problem of using machine learning for flaky test detection.

4.5.5 Implications

Researchers. This study shows that using FLAKE16 improved machine learning-based flaky test detection performance compared to the FLAKEFLAGGER dataset. This demonstrates that measuring a greater diversity of test case properties allows classifiers to better distinguish between flaky and non-flaky test cases. The results also reveal that machine learning classifiers are just as applicable to detecting OD flaky tests as they are to detecting NOD flaky tests. Therefore, researchers should consider how machine learning classifiers can improve the scalability of OD flaky test detection, since many previous techniques incur a significant time cost [47, 90, 175].

Developers. This study establishes that the maximum number of concurrently running threads during test case execution is a very impactful feature when detecting NOD flaky tests. As such, my advice to developers would be to avoid concurrency in tests as much as is possible. When developers cannot heed this advice, it may be useful for them to assume such tests are likely to be flaky [68]. The same can be said for test cases that perform significant input and output, given that I found the number of read- and write- related system calls to be impactful features for detecting both NOD and OD flaky tests.

4.6 Related Work

Lam et al. [90] presented IDFLAKIES, a technique for detecting flaky tests and labelling them as OD or NOD. Initially, IDFLAKIES repeatedly executes a test suite in its *original* test run order, the default order scheduled by the test runner, to identify which test cases pass consistently. It then repeatedly executes the test suite in *modified* orders to distinguish between NOD and OD flaky tests. Since IDFLAKIES requires many repeated test executions, it may not scale well to large or slow-running test suites. Bell et al. [16] presented DEFLAKER, which, unlike IDFLAKIES, cannot identify OD flaky tests. Should a test case fail, having passed on a previous version of the software under test and without covering any modified code, DEFLAKER labels it as flaky. To measure coverage, DEFLAKER requires instrumentation. Likewise, Flake16Framework requires instrumentation to measure coverage and other metrics in FLAKE16. In both cases, this instrumentation introduces run-time overhead. However, Flake16Framework only requires a single instrumented run to detect flaky tests, whereas DEFLAKER requires instrumentation every time it is used.

The drawbacks of previous techniques, specifically the high volume of test executions, motivated several studies to evaluate machine learning classifiers for detecting flaky tests. Bertolino et al. [159] presented FLAST for predicting if a test case is flaky based purely on its source code.

Their technique uses a *k-nearest neighbour* classifier [80] which labels test cases based on their cosine distance to labelled training instances within a *bag-of-words* feature space. The bag-of-words approach is used to represent test cases as sparse vectors where each element corresponds to the frequency of a particular identifier or keyword in its source code. Pinto et al. [128] performed a similar study but included additional static features beyond the bag-of-words representation such as the number of lines of code that make up a test case. This work was subsequently replicated and expanded by Haben et al. [65]. Alshammari et al. [8] presented FLAKEFLAGGER, a Random Forest classifier encoding test cases with a feature set that is mostly a subset of FLAKE16. Their evaluation showed that their feature set offered a 347% improvement in overall F1 score compared to Pinto et al.’s purely static feature set at the relatively minimal cost of the single test suite run, required to collect the dynamic features. One aspect these three studies have in common is that they used datasets based on Bell et al.’s evaluation of DEFLAKER [16]. Recall that DEFLAKER does not detect OD flaky tests, meaning they would be labelled as *non-flaky* during the training and evaluation of the classifiers in these studies.

4.7 Conclusion

This study presented FLAKE16, a new feature set that encodes test cases for machine learning-based flaky test detection. I evaluated the performance of 54 machine learning pipelines when detecting both NOD and OD flaky tests using both FLAKE16 and a previously established feature set. For both categories of flaky test, experiments involving 26 real-world Python projects showed greater detection performance when using FLAKE16. Offering a more complete evaluation of the problem of flaky test detection, this study is the first to apply machine learning classifiers to the detection of OD flaky tests. Using the SHAP technique to evaluate the impact that each FLAKE16 feature has on the classifiers’ decisions, the results show the peak number of concurrently running threads during test case execution to be the most impactful for detecting NOD flaky tests. For OD flaky tests, the number of read- and write-related system calls have the greatest impact. The experiments also reveal that static code complexity features such as cyclomatic complexity, Halstead volume, and maintainability has little impact in both cases.

As demonstrated by Table 4.4, the machine learning pipelines were very fast but their detection performance was suboptimal, with F1 scores around 0.5. This raises the question of how machine learning classifiers can be combined with rerunning-based detection techniques. Such a hybrid approach could potentially outperform a classifier while imposing a time cost that is significantly lower than rerunning every test case. Arguably, this would have considerably more utility with respect to mitigating flaky tests because it is accurate enough to identify the majority of the flaky tests in a test suite but fast enough that developers can realistically be expected to use it.

Chapter 5

Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models

The contents of this chapter is based on “O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(72), 2023”.

5.1 Introduction

The research community has introduced a multitude of automated techniques to detect flaky tests. Many are *rerunning-based*, meaning they may require an excessive number of repeated test case executions, making them expensive for deployment in large software projects [90, 175]. Alshammari et al. [8] repeatedly executed the test suites of 24 Java projects and were still detecting non-order-dependent (NOD) flaky tests after 10,000 reruns. The time cost of rerunning-based detection led researchers to investigate techniques that do not require test case runs but are instead based on machine learning classifiers [8, 128, 159]. However, they only offer an approximate solution in this domain. For example, in Chapter 4, after evaluating 54 machine learning pipelines for detecting flaky tests, the best detection performance I observed was an F1 score of around 0.5. The prohibitive time cost of rerunning-based techniques and the limited performance of machine learning-based techniques leaves practitioners with a stark choice.

This study introduces CANNIER (maChine leArNiNg assIsted tEst Rerunning), a high-level approach for reducing the time cost of rerunning-based detection techniques by combining them with machine learning classifiers. It does this by using the output of the classifier as a heuristic to reduce the problem space for the rerunning-based technique. I demonstrate the applicability of CANNIER by instantiating it for three previously established detection techniques. I implemented these within an automated tool and empirically evaluated them using 30 Python projects as subjects. I found that CANNIER could significantly reduce time cost at the expense of only a minor reduction in detection performance. For example, by applying CANNIER to the Classification stage of IDFLAKIES [90] (that distinguishes NOD flaky tests from OD flaky tests), I was able to reduce its time cost by 84% at the expense of misclassifying just 8 flaky tests out of 1,130. Therefore, CANNIER represents a “best of both worlds” solution to flaky test detection. Furthermore, the evaluation of CANNIER builds upon the findings from Chapter 4 by extending the FLAKE16 feature set, evaluating the impact of repeated feature measurements on detection performance, and analysing the impact of each feature in greater detail.

In summary, the main contributions of this study are:

1. **Approach (Section 5.3):** CANNIER significantly reduces the time cost of rerunning-based flaky test detection with a minimal decrease in detection performance.
2. **Tooling (Section 5.4):** To facilitate the empirical evaluation and allow for replication, I developed a framework of automated tools that I make freely available [223].
3. **Evaluation (Section 5.5):** A comprehensive empirical evaluation demonstrated the effectiveness of CANNIER’s combination of rerunning and machine learning techniques, revealing further novel findings about machine learning-based flaky test detection, such as the performance of machine learning classifiers for detecting polluter test cases.
4. **Dataset (Section 5.5.1):** A dataset containing 89,668 tests from 30 Python projects taking over six weeks of compute time to produce. I make this available as part of the replication package to enable further research [217].

5.2 Background

5.2.1 RERUN

The research community has presented many automated flaky test detection techniques that are based on rerunning test cases. The most straight-forward such technique is to repeatedly execute a test case until it exhibits both passing and failing behaviour. In its most basic form, this technique involves rerunning the test cases of a test suite in the same test run order and under the same environmental conditions each time [16]. I refer to this specific technique as RERUN. Since the test run order remains constant, RERUN can only identify non-order-dependent (NOD) flaky tests. As its only parameter, RERUN requires an upper-limit on the number of times to execute a test case without observing an inconsistent outcome. If the upper-limit is reached, the technique classifies the test case as non-flaky and stops rerunning it. Since many test cases may require hundreds or even thousands of runs to manifest their flakiness (see Figure 4.1), this technique can become very expensive for long-running test suites, thus limiting the technique in practice.

5.2.2 IDFLAKIES

Lam et al. [90] presented IDFLAKIES, a technique for detecting flaky tests and classifying them as NOD or a victim¹ (OD). The technique consists of three stages: *Setup*, *Running*, and *Classification*. In the Setup stage, IDFLAKIES repeatedly executes the test suite in its original order to identify and filter any consistently-failing test cases. In the Running stage, IDFLAKIES continues to rerun the test suite, but this time in modified test run orders. In the Classification stage, for every test case that failed during the Running stage, IDFLAKIES re-executes the test suite in both the original order and in the modified order that witnessed the failure, truncated up to and including the failing test case. I refer to this stage as IDFCCLASS (IDFLAKIESCLASSIFICATION). Should the test case fail again in the truncated modified order and pass again in the truncated original order, IDFLAKIES classifies it as a victim. Otherwise, it classifies the test case as NOD. Should a test case fail multiple times during the Running stage, IDFLAKIES can repeat the Classification stage for a percentage of the additional failures for greater confidence in the final label.

IDFLAKIES has several parameters: the number of reruns during the setup stage, the number of reruns during the Running stage, the method of generating the modified test run orders during the Running stage (e.g., shuffle), and the percentage of additional failures to recheck in the Classification stage. Depending on the choice of values for these parameters, IDFLAKIES can require a significant number of test executions and thus impose a prohibitive time cost.

¹The paper that introduced the victim/polluter terminology [141] also introduced the terms *brittle/state-setter* for when the order-dependent flaky test fails in isolation. In this chapter, I use the victim/polluter terms generally, regardless of the order-dependent flaky test’s outcome.

5.2.3 PAIRWISE

While the IDFLAKIES technique can detect victim flaky tests, it cannot identify their associated polluters. Polluters are the test cases that induce order-dependency in victims by leaving behind side-effects [141]. Zhang et al. [175] proposed a technique that can detect a subset of a test suite’s victims and their polluters (although the authors designed the technique primarily for detecting victims). It involves executing every permutation of test cases of length two (every pair in both orders) in isolation, such as in separate Java Virtual Machine or Python interpreter processes. I refer to this technique as PAIRWISE. Initially, PAIRWISE requires an expected outcome for every test case. It could obtain these by executing each test case in isolation to observe their outcome independent of the possible side-effects of other test cases.

Once every test case has an expected outcome, PAIRWISE executes every 2-permutation of test cases, such that each test case has a turn at being both the first and second to be executed in the pair — the candidate polluter and victim, respectively. For more reliable results, PAIRWISE ought to filter out any pairs with a known NOD flaky test as the candidate victim because they do not have a reliable expected outcome (although Zhang et al. did not propose this filtering stage in their paper). For a given pair, if the second test case yields an outcome different from expected, PAIRWISE classifies it as a victim and classifies the first test as one of its polluters.

Previous work has determined that an order-dependency can involve more than two test cases [141], though as part of their empirical study, Zhang et al. found that 76% of order-dependencies did involve just two. Considering only pairs of test cases, the time complexity of PAIRWISE is already quadratic in the size of the test suite and hence very expensive, and so to consider longer permutations would quickly render the technique intractable.

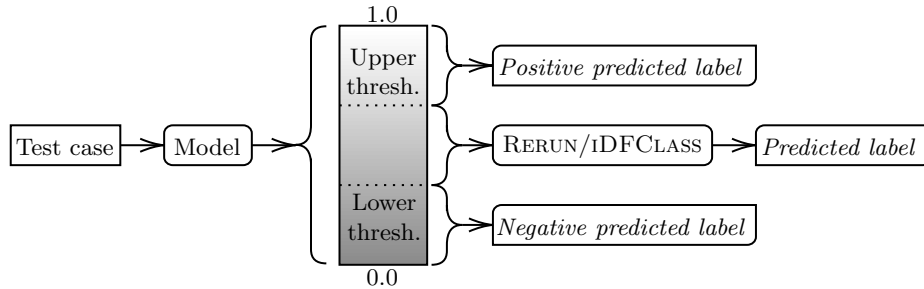
5.3 The CANNIER Approach

CANNIER (maChine leArNiNg assIsted tEst Rerunning) is a high-level approach that combines a rerunning-based flaky test detection technique and one or more machine learning classifiers. The classifiers must provide a predicted probability that a given test case is flaky. The general concept behind CANNIER is to use the predicted probabilities as a heuristic to reduce the problem space for the rerunning-based technique. As attested by the later empirical evaluation (see Section 5.5), this approach can dramatically reduce the number of test case executions, and therefore time cost, at the expense of only a minor decrease in detection performance. The specifics of how CANNIER uses the predicted probabilities depends on the nature of the rerunning-based technique. Figure 5.1 provides a visual summary of the application of CANNIER to the three rerunning-based detection techniques introduced in Section 5.2.

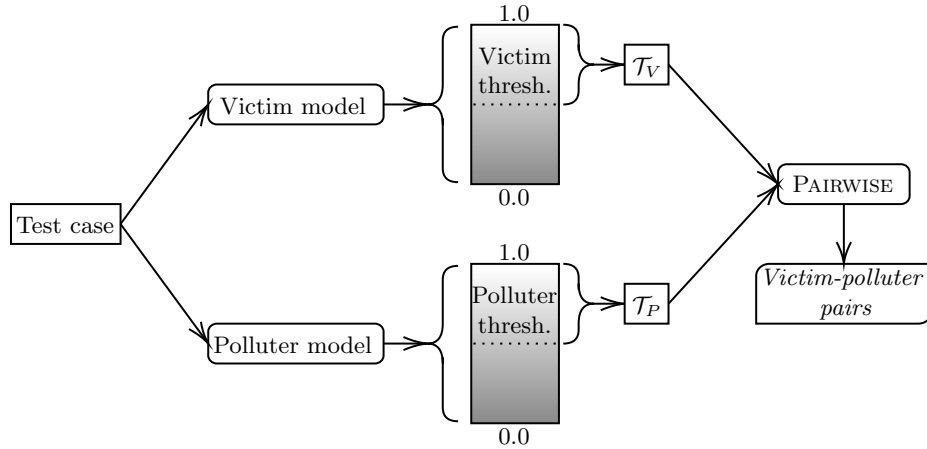
5.3.1 Motivating Example

I used the `airflow` project, developed by the Apache Software Foundation, as one of the subjects in the empirical evaluation [201]. Its test suite contains 3,251 test cases as of version 1.10.14. I executed the test suite 2,500 times in its original order and identified 66 NOD flaky tests. Following the empirical evaluation, I found that the single-core time cost to detect these flaky tests, using RERUN with a maximum of 2,500 reruns per test case, is 1.69×10^6 seconds. This is based on the time cost of each individual test case that I measured on a machine with a 24-core AMD Ryzen 5900X CPU. Having the same number of virtual cores and a comparable single-core performance, `m5zn.6xlarge` is arguably the most similar cloud instance offered by Amazon Web Services [257]. As of August 2022, Amazon offers this instance at the on-demand hourly rate of 1.982 USD. This means that to detect the NOD flaky tests in `airflow` using RERUN would take $((1.69 \times 10^6) \div 24) \div 60^2 \approx 19.56$ hours and cost $19.56 \times 1.982 \approx 38.77$ USD on this instance.

Given the cost in both time and money of using RERUN to detect flaky tests, a developer may instead opt to use a machine learning classifier. I trained an Extra Trees classifier, a variation of Random Forest [51], to detect NOD flaky tests and evaluated it using stratified 10-fold cross



(a) For RERUN and iDFCLASS, CANNIER uses a single machine learning classifier to predict the probability of a given test case being of the positive class (flaky). If the probability is below a lower-threshold or above an upper-threshold, CANNIER assigns the test case a negative or positive predicted label respectively. In the ambiguous region between the two thresholds, CANNIER delegates to the rerunning-based technique to predict the label.



(b) For PAIRWISE, CANNIER uses one machine learning classifier to predict the probability of each test case being a victim and another to do the same for being a polluter. All those with a probability of being a victim above a threshold join the set of candidate victims, \mathcal{T}_V . Similarly, those with a probability of being a polluter above a threshold enter the candidate polluters set, \mathcal{T}_P . CANNIER then restricts PAIRWISE to consider only test cases from \mathcal{T}_P as the first test case of each pair, and only test cases from \mathcal{T}_V as the second.

Figure 5.1: CANNIER uses the predicted probabilities from one or more machine learning classifiers as a heuristic to reduce the problem space of a rerunning-based flaky test detection technique. A single machine learning classifier is suitable for RERUN and the Classification stage of iDFLAKIES (iDFCLASS) (a). Two machine learning classifiers are required for PAIRWISE (b).

validation. Within `airflow`, I found that it misclassified 26 test cases that were flaky as non-flaky, and 26 test cases that were non-flaky as flaky. With only 40 of the 66 NOD flaky tests cases actually classified as such, the classifier achieved a precision of $40 \div (40 + 26) \approx 61\%$ and a recall of $40 \div (40 + 26) \approx 61\%$. In this context, precision is the percentage of detected flaky tests that are genuinely flaky and recall is the percentage of genuinely flaky tests that were detected. Therefore, machine learning-based detection offers a very approximate solution. Because the classifier uses dynamic features, the time cost of applying it is approximately equal to the time cost of a single test suite run to produce a feature vector for each test case. I observed a single-core time cost for this of 7.77×10^2 seconds for `Airflow`. This would require $((7.77 \times 10^2) \div 24) \div 60^2 \approx 0.01$ hours on the `m5zn.6xlarge` instance, costing $0.01 \times 1.982 \approx 0.02$ USD. I do not consider the time cost associated with applying the Extra Trees classifier to each test case because it is negligible relative to the time taken to execute the test suite (typically less than one second, see Table 4.4). I also do not consider the time taken to train the Extra Trees classifier. This is because the classifier only needs to be trained once and can then be applied any number of times, and so I consider training to be an off-line stage that does not contribute to the time cost of applying the classifier.

Rerunning-based detection and machine learning-based detection represent opposite extremes. As shown in this example, `RERUN` is very expensive and the Extra Trees classifier is cheap but very approximate. By applying `CANNIER` to `RERUN` (`CANNIER+RERUN`), developers get a flaky test detection technique that is much cheaper than `RERUN` and much more accurate than the Extra Trees classifier. Following the empirical evaluation, I found that the single-core time cost to detect the 66 NOD flaky tests in `airflow` using `CANNIER+RERUN` is 7.71×10^5 seconds. This would require $((7.71 \times 10^5) \div 24) \div 60^2 \approx 8.92$ hours on an `M5ZN.6XLARGE` instance at a cost of $8.92 \times 1.982 \approx 17.68$ USD. Therefore, `CANNIER` reduces the cost in USD of `RERUN` by 54%. I also found that it misclassified three flaky tests as non-flaky but correctly classified the remaining 62. This leads to a precision of $63 \div (63 + 0) = 100\%$ and a recall of $63 \div (63 + 3) \approx 95\%$. This is far more accurate than the Extra Trees classifier that only achieved a precision and recall of 61%.

The empirical evaluation demonstrates that `CANNIER` is effective for multiple projects and the three rerunning-based detection techniques introduced in Section 5.2. For the whole dataset of 89,668 test cases from 30 projects, I found that `CANNIER` was able to reduce the time cost (and therefore monetary cost) by an average of 88% across the three techniques.

5.3.2 Single-Classifier CANNIER

Using `CANNIER` with a single machine learning classifier is suitable for reducing the time cost of `RERUN` and `IDFCLASS`. In the case of `RERUN`, the flaky test classification problem is that of distinguishing NOD flaky tests from the rest of the test cases. Since it is a binary problem, NOD flaky tests are the positive class and the rest of the test cases are the negative. For `IDFCLASS`, it is telling apart NOD and victim (OD) flaky tests. In this case, NOD flaky tests are the positive class and victims are the negative. For both `RERUN` and `IDFCLASS`, the machine learning classifier should provide a predicted probability of belonging to the positive class for each test case. `CANNIER` assigns a positive predicted label to a test case if this probability is above an upper threshold and a negative predicted label if it is below a lower threshold. This leaves an ambiguous region between the two thresholds. `CANNIER` delegates any test cases with predicted probabilities within this ambiguous region to the rerunning-based technique.

5.3.3 Multi-Classifier CANNIER

Using two classifiers, `CANNIER` can reduce the time cost of `PAIRWISE`. The first classifier is used to predict the probability of each test case being a victim. In other words, it addresses the classification problem of distinguishing victims from non-victims. The second is used to do the same but for being a polluter. In both instances, `CANNIER` classifies every test case above a threshold as the positive class (a victim or a polluter) and every other test case as the negative (not a victim or not a polluter). In this way, `CANNIER` produces two non-mutually exclusive sets, one of victims, \mathcal{T}_V , and one of polluters, \mathcal{T}_P (there is no reason why a test case cannot be both a

Table 5.1: The 18 features measured by `pytest-CANNIER`.

# Feature	Description
1 Read Count	Number of times the filesystem had to perform input [241].
2 Write Count	Number of times the filesystem had to perform output [241].
3 Run Time	Elapsed wall-clock time of the whole test case execution.
4 Wait Time	Elapsed wall-clock time spent waiting for input/output operations.
5 Context Switches	Number of voluntary context switches.
6 Covered Lines	Number of lines covered.
7 Source Covered Lines	Number of lines covered that are not part of test cases.
8 Covered Changes	Number of times each covered line has been modified in the last 75 commits.
9 Max. Threads	Peak number of concurrently running threads.
10 Max. Children	Peak number of concurrently running child processes.
11 Max. Memory	Peak memory usage.
12 AST Depth	Maximum depth of nested program statements in the test case code.
13 Assertions	Number of assertion statements in the test case code.
14 External Modules	Number of non-standard modules (i.e., libraries) used by the test case.
15 Halstead Volume	A measure of the size of an algorithm’s implementation [7, 126, 129].
16 Cyclomatic Complexity	Number of branches in the test case code [52, 126, 129].
17 Test Lines of Code	Number of lines in the test case code [126, 129].
18 Maintainability	A measure of how easy the test case code is to support and modify [164, 279].

victim and a polluter [162]). Then, CANNIER applies PAIRWISE with only the members of \mathcal{T}_P as the first test in each pair and only the members of \mathcal{T}_V as the second. Therefore, CANNIER can reduce the time complexity of PAIRWISE from $O(|\mathcal{T}|^2)$, where \mathcal{T} is the set of all test cases in the test suite, to $O(|\mathcal{T}_V| \times |\mathcal{T}_P|)$, that is considerably faster even when \mathcal{T}_V and \mathcal{T}_P are not significantly smaller than \mathcal{T} .

5.4 Tooling

To produce the dataset and facilitate the empirical evaluation, I developed a suite of automated tools including a plugin for the Python testing framework `pytest` [266], named `pytest-CANNIER` [225], and a command-line tool named `CANNIER-Framework` [223]. The purpose of `pytest-CANNIER` is to add the functionality to `pytest` necessary for the evaluation. This includes recording test case outcomes and measuring feature values. The purpose of `CANNIER-Framework` is to automate every aspect of the evaluation, including executing `pytest-CANNIER` on the subject test suites, collating raw data, and training and evaluating machine learning classifiers.

5.4.1 `pytest-CANNIER`

I decided to target `pytest` due to its compatibility with test suites written for other frameworks such as `unittest` [278]. `pytest-CANNIER` takes a test suite \mathcal{T} as input and offers four execution modes: *Baseline*, *Shuffle*, *Features*, and *Victim*. In the *Baseline* mode, the plugin executes the test suite as normal. For each test case $t \in \mathcal{T}$, `pytest-CANNIER` records its outcome b_t , that is either pass, $b_t = 0$, or fail, $b_t = 1$. In the *Shuffle* mode, the plugin randomises the order of the test cases and records the outcome of every test case s_t .

In the *Features* mode, `pytest-CANNIER` produces a feature vector $\mathbf{x}_t \in \mathbb{R}^{18}$, for each test case t . This contains the 16 features of FLAKE16 alongside two additional metrics. The first of these is *Wait Time*. This is the amount of time during test case execution spent waiting for input/output (I/O) operations to complete. Previous research identified I/O in test cases as being potentially associated with flakiness [105]. The second additional feature is *Max. Children*. This measures the peak number of concurrently running child processes. A finding that many empirical studies have in common is that asynchronous operations and concurrency are very frequent causes of flaky

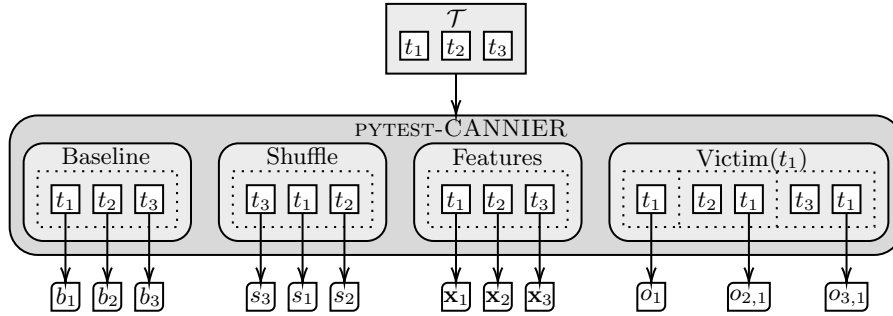


Figure 5.2: As input, `pytest-CANNIER` takes a test suite $\mathcal{T} = (t_1, t_2, t_3)$ and can be launched in four modes: *Baseline*, *Shuffle*, *Features*, or *Victim*. In the *Baseline* mode, the plugin runs the test suite in its original order and records the pass/fail outcome of every test case (b_1, b_2, b_3). In the *Shuffle* mode, `pytest-CANNIER` executes the test suite in a random order and also records test case outcomes (s_1, s_2, s_3). In the *Features* mode, the plugin produces a feature vector for each test case ($\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$). In the *Victim* mode, `pytest-CANNIER` takes a victim test case as an additional input (t_1) and initially executes it in isolation to ascertain its expected outcome (o_1). Then, the plugin executes every other test case in a separate process with the victim immediately following and records its outcome ($o_{2,1}, o_{3,1}$). This is to identify polluters of the victim.

tests [37, 89, 105, 136]. This was my rationale for the inclusion of *Max. Threads* into FLAKE16. However, due to the global interpreter lock implemented within the CPython interpreter [226], it may be necessary for developers to achieve concurrency with child processes. Table 5.1 offers a description of all 18 features. In the *Victim* mode, the plugin takes a test case v , executes the test sequence $\langle v \rangle$, and records the outcome of v , o_v . This is to ascertain the expected outcome of v when executed in isolation from the rest of the test suite. Following this, `pytest-CANNIER` executes the sequences $\langle p, v \rangle$ for every test case p in $\mathcal{T} - \{v\}$, while recording the outcome of v when executed immediately after each p , $o_{p,v}$. This is to identify the polluters of v where $o_{p,v} \neq o_v$. For isolation between sequence runs, the plugin executes them in separate Python processes [16, 175]. This implements the PAIRWISE technique with respect to a single candidate victim v . Figure 5.2 provides a visual summary of `pytest-CANNIER`.

5.4.2 CANNIER-Framework

Classifier Training and Evaluation Data

As input, `CANNIER-Framework` takes a subject set of test suites \mathcal{U} . With every test suite $\mathcal{T} \in \mathcal{U}$ as input, the framework executes the plugin N_B times in the *Baseline* mode, resulting in N_B values of b_t , $(b_{t,1}, b_{t,2}, \dots, b_{t,N_B})$, for each test case $t \in \mathcal{T}$. Similarly, `CANNIER-Framework` runs every test suite N_S times in the *Shuffle* mode, leading to N_S values of s_t , $(s_{t,1}, s_{t,2}, \dots, s_{t,N_S})$. The framework counts the number of times that every test case fails in the *Baseline* mode B_t and in the *Shuffle* mode S_t . The definition of both values is given in the following equation.

$$B_t = \sum_{i=1}^{N_B} b_{t,i} \quad S_t = \sum_{i=1}^{N_S} s_{t,i} \quad (5.1)$$

`CANNIER-Framework` also executes each test suite N_F times with `pytest-CANNIER` in the *Features* mode, resulting in N_F feature vectors for every test case t ($\mathbf{x}_{t,1}, \mathbf{x}_{t,2}, \dots, \mathbf{x}_{t,N_F}$). As an additional input, the framework takes \mathcal{I} , a random sample of n_F indices ranging from 1 to N_F inclusive without replacement. With this, the framework produces a mean feature vector $\mathbf{X}_t(\mathcal{I})$, to encode each test case according to the following equation.

$$\mathbf{X}_t(\mathcal{I}) = \frac{1}{n_F} \sum_{i \in \mathcal{I}} \mathbf{x}_{t,i} \quad (5.2)$$

Table 5.2: The four flaky test classification problems. For test suite \mathcal{T} , the domain \mathcal{T}_ϕ is the subset of \mathcal{T} that is relevant to the problem ϕ . For a specific \mathcal{T}_ϕ , the negative class for each problem (\mathcal{T}_ϕ^-) is the complement of the positive.

ϕ	Name	Domain (\mathcal{T}_ϕ)	Positive Class (\mathcal{T}_ϕ^+)
1	NOD-vs-Rest	\mathcal{T}	$\{t t \in \mathcal{T}_\phi, 0 < B_t < N_B\}$
2	NOD-vs-Victim	$\{t t \in \mathcal{T}, B_t < N_B \wedge S_t > 0\}$	$\{t t \in \mathcal{T}_\phi, 0 < B_t < N_B\}$
3	Victim-vs-Rest	\mathcal{T}	$\{t t \in \mathcal{T}_\phi, (B_t = 0 \vee B_t = N_B) \wedge B_t \neq S_t\}$
4	Polluter-vs-Rest	\mathcal{T}	$\{p p \in \mathcal{T}_\phi, \exists v \in \mathcal{T}_\phi - \{p\} (o_{p,v} \neq o_v)\}$

For each $\mathcal{T} \in \mathcal{U}$, the framework runs the Victim mode of `pytest-CANNIER` with every test case that had a consistent outcome in the Baseline mode ($B_v = 0 \vee B_v = N_B$) and an inconsistent outcome in the Shuffle mode ($B_v \neq S_v$) as the candidate victim v . The former condition is to ensure that every v has the reliable expected outcome that PAIRWISE requires. The latter is a time saving measure — if a test case is consistent in the Shuffle mode then it is very unlikely to be a victim and therefore would have no polluters. For the purposes of greater reproducibility and isolation, `CANNIER-Framework` executes the plugin in a separate Docker container for every run of a test suite [214]. The Dockerfile contains all the commands needed to reproduce the Docker image and is available as part of the replication package [217].

Once the plugin has finished performing the test suite runs, `CANNIER-Framework` determines a *ground-truth label* $y_{t,\phi}$, for every test case t in the whole subject set, $t \in \bigcup_{\mathcal{T} \in \mathcal{U}} \mathcal{T}$, and flaky test classification problem ϕ . Recall from Section 5.3 that these problems are: NOD flaky tests versus the rest of the test cases (NOD-vs-Rest, $\phi = 1$), NOD flaky tests versus victim flaky tests (NOD-vs-Victim, $\phi = 2$), victim flaky tests versus the rest (Victim-vs-Rest, $\phi = 3$), and polluters versus the rest (Polluter-vs-Rest, $\phi = 4$). Each problem has a domain $\mathcal{T}_\phi \subseteq \mathcal{T}$, that is the subset of test cases in a given test suite \mathcal{T} that are relevant. Since the problems are binary classifications, they also have a positive class, $\mathcal{T}_\phi^+ \subset \mathcal{T}_\phi$, and a negative class, $\mathcal{T}_\phi^- = \mathcal{T}_\phi - \mathcal{T}_\phi^+$. The ground-truth label for a test case is positive if it is in the positive class of a problem ($y_{t,\phi} = 1$) and negative otherwise ($y_{t,\phi} = 0$). For a test case t belonging to test suite \mathcal{T} , the following equation defines the ground truth label $y_{t,\phi}$.

$$y_{t,\phi} = \begin{cases} 0 & \text{if } t \in \mathcal{T}_\phi^- \\ 1 & \text{if } t \in \mathcal{T}_\phi^+ \end{cases} \quad (5.3)$$

For the NOD-vs-Rest problem ($\phi = 1$), the positive class is the set of NOD flaky tests, that I define as those with an inconsistent outcome in the Baseline mode ($0 < B_t < N_B$). The only test cases that are relevant to the NOD-vs-Victim problem ($\phi = 2$) are those that did not consistently fail during the runs in the Baseline mode ($B_t < N_B$) and failed at least once in Shuffle mode ($S_t > 0$). The former condition corresponds to the Setup stage of IDFLAKIES where such test cases would be excluded from further analysis. The latter corresponds to the Running stage, where any test case that fails at least once goes on to the Classification stage. For this problem, the positive class is also the set of NOD flaky tests. For the Victim-vs-Rest problem ($\phi = 3$), the positive class is the set of test cases with a consistent outcome in the Baseline mode ($B_t = 0 \vee B_t = N_B$) and an inconsistent outcome in the Shuffle mode ($B_t \neq S_t$). This represents the set of victims. Finally, for the Polluter-vs-Rest problem ($\phi = 4$), the positive class is the set of test cases that behaved as polluters in the Victim mode. Table 5.2 gives a definition of each problem.

Classifier Training and Evaluation Procedure

`CANNIER-Framework` follows a general machine learning pipeline for classifier training and evaluation. The pipeline leaves the specific classifier and data balancing technique unspecified, such that it can be instantiated with a choice for both of these components to create a concrete pipeline. The pipeline performs stratified 10-folds cross validation. This creates ten *folds* where 90% of the test cases in the whole subject set are for training and the other 10% are for evaluation. The

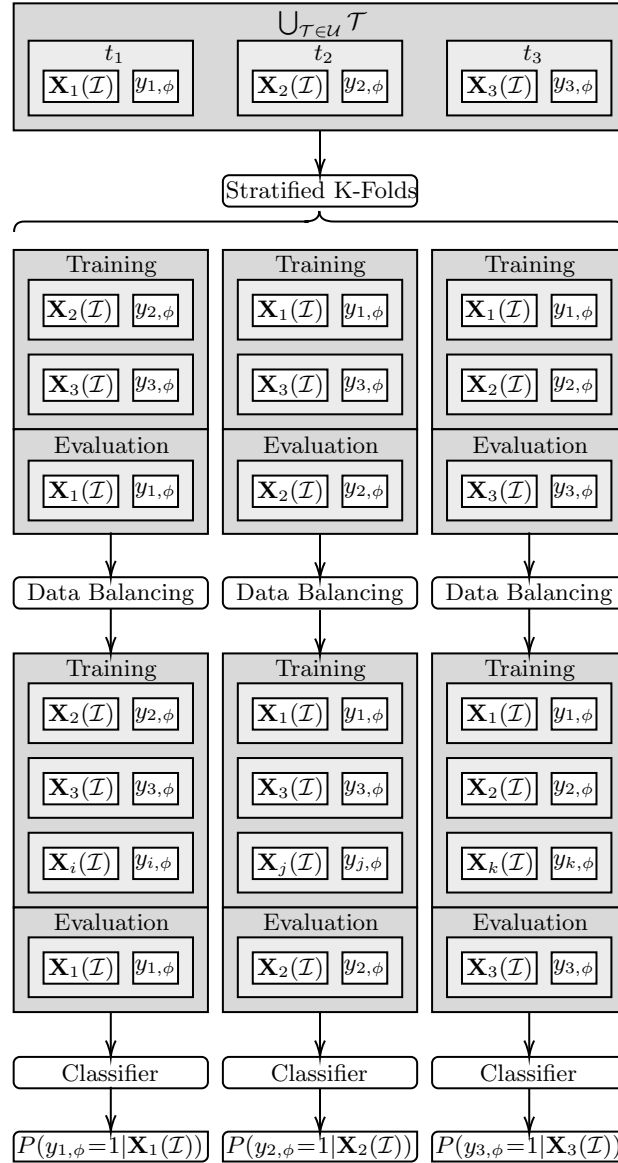


Figure 5.3: **CANNIER-Framework** performs stratified k-folds cross validation upon the set of all test cases in the subject set, $\bigcup_{\mathcal{T} \in \mathcal{U}} \mathcal{T}$. Following this, it applies a data balancing technique to the training portion of each fold. The framework then trains a machine learning classifier using the mean feature vectors $\mathbf{X}_t(\mathcal{I})$, and ground-truth labels $y_{t,\phi}$, of every test case t in each training portion. Finally, for each fold, **CANNIER-Framework** applies the trained classifier to the feature vectors of every test case in the evaluation portion. Since the evaluation portion of each fold is unique, every test case ends up with a predicted probability of being in the positive class, $P(y_{t,\phi}=1|\mathbf{X}_t(\mathcal{I}))$.

class proportion of each fold roughly follows that of the whole subject set, and since that is highly imbalanced for every classification problem, the framework applies the data balancing technique to the training set only [23]. For each fold, the framework fits the machine learning classifier with the training set and applies it to every test case in the evaluation set. For a given problem ϕ , this results in a *predicted probability* $P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I}))$, of each test case in the evaluation set being of the positive class. Since the evaluation portion of every fold is unique, after ten folds each test case in the whole subject set has a prediction. Figure 5.3 offers an overview of the general pipeline. Given a lower-threshold ω_l and an upper-threshold ω_u on the predicted probability as further inputs, **CANNIER-Framework** assigns a *predicted label* $z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)$, to every test case, as previously shown in Figure 5.1. The following equation defines the predicted label for a test case, denoted $z_{t,\phi}$.

$$z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u) = \begin{cases} 0 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) < \omega_l \\ 1 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ y_{t,\phi} & \text{if } \omega_l \leq P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) < \omega_u \end{cases} \quad (5.4)$$

Using the ground-truth and predicted labels for each test case in a given test suite \mathcal{T} , the framework calculates the frequencies of the four confusion matrix categories: true-positive (TP), false-positive (FP), false-negative (FN), and true-negative (TN). From these, it calculates the Matthews correlation coefficient (MCC) to assess the detection performance of the machine learning classifier for a given problem ϕ . The possible values of MCC are the closed real range between -1 and 1, where 1 indicates a classifier with perfect agreement between the ground-truth labels and the predicted labels and 0 indicates a classifier that is no better than random guessing of the predicted labels. A classifier with an MCC of -1 indicates perfect disagreement between the ground-truth labels and the predicted labels, such that taking a classifier with an MCC of 1 and inverting the predicted labels would yield an MCC of -1. I selected MCC as the overall performance metric, as opposed to F1 score, because it only produces a high value if the classifier performs well in terms of all four confusion matrix categories, whereas F1 score ignores true-negatives [24]. See Figure 5.4 for a summary of how **CANNIER-Framework** combines **pytest-CANNIER** and the general machine learning pipeline from Figure 5.3 to produce this data. The following equation defines $MCC_{\mathcal{T}_\phi}$ with respect to the four confusion matrix categories respectively denoted as $TP_{\mathcal{T}_\phi}$, $FP_{\mathcal{T}_\phi}$, $FN_{\mathcal{T}_\phi}$, and $TN_{\mathcal{T}_\phi}$.

$$\begin{aligned} TP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} y_{t,\phi} z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u), \\ FP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} [1 - y_{t,\phi}] z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u), \\ FN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} y_{t,\phi} [1 - z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)], \\ TN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} [1 - y_{t,\phi}] [1 - z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)], \\ MCC_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \frac{TP_{\mathcal{T}_\phi} TN_{\mathcal{T}_\phi} - FP_{\mathcal{T}_\phi} FN_{\mathcal{T}_\phi}}{\sqrt{(TP_{\mathcal{T}_\phi} + FP_{\mathcal{T}_\phi})(TP_{\mathcal{T}_\phi} + FN_{\mathcal{T}_\phi})(TN_{\mathcal{T}_\phi} + FP_{\mathcal{T}_\phi})(TN_{\mathcal{T}_\phi} + FN_{\mathcal{T}_\phi)}} \end{aligned} \quad (5.5)$$

Technique Evaluation Procedure

CANNIER-Framework evaluates the application of CANNIER to RERUN (**CANNIER+RERUN**), the Classification stage of IDFLAKIES (**CANNIER+iDFCLASS**), and PAIRWISE (**CANNIER+PAIRWISE**). I developed a mathematical model, that I implemented within the framework, to estimate the detection performance and single-core time cost associated with a set of parameters for the three techniques. **CANNIER-Framework** uses the ground-truth labels and predicted probabilities for each test from the NOD-vs-Rest problem ($\phi = 1$) to model **CANNIER+RERUN**. It uses the data from the NOD-vs-Victim problem ($\phi = 2$) to model **CANNIER+iDFCLASS** in an equivalent fashion. In both of these cases, the ground-truth labels represent the output from the

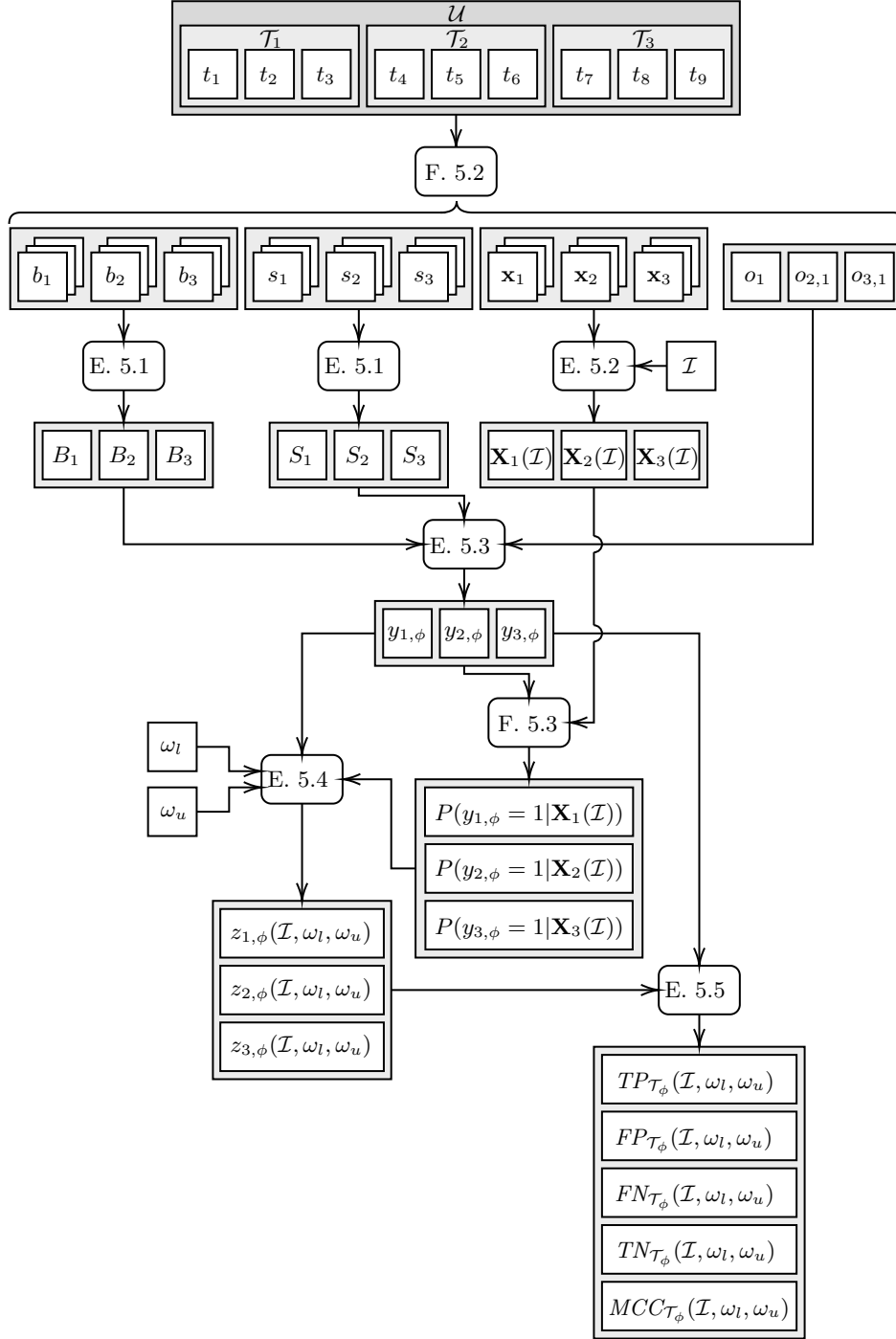


Figure 5.4: An overview of how CANNIER-Framework combines pytest-CANNIER and the general machine learning pipeline, with subject set \mathcal{U} , random sample \mathcal{I} , and thresholds ω_l and ω_u as input. It references previously defined figures (F.) and equations (E.).

“RERUN/IDFCCLASS” block and the predicted probabilities represent the output from the “Model” block in Figure 5.1a. The parameters of CANNIER+RERUN are the lower- and upper-thresholds on the classifier prediction, ω_l and ω_u , the sample size to produce the mean feature vectors for each test case, denoted n_F , and the maximum number of times to execute a test case without observing an inconsistent outcome, written as R_{max} . For CANNIER+IDFCCLASS, the parameters are ω_l , ω_u , n_F , and the percentage of additional failures to recheck, denoted γ . The framework uses the outcomes from the Victim mode and the predicted probabilities from the Victim-vs-Rest ($\phi = 3$) and Polluter-vs-Rest ($\phi = 4$) problems to model CANNIER+PAIRWISE. The outcomes represent the “PAIRWISE” block and the predicted probabilities represent the “Victim model” and “Polluter model” blocks in Figure 5.1b. For CANNIER+PAIRWISE, the parameters are the threshold for the victim classifier ω_V , the threshold for the polluter classifier ω_P , and n_F .

Given a random sample \mathcal{I} of size n_F along with ω_l and ω_u , CANNIER-Framework estimates the detection performance of CANNIER+RERUN and CANNIER+IDFCCLASS as an MCC value. For every test case t in a given test suite \mathcal{T} , the framework needs its individual time cost C_t , and the number of times RERUN is expected to execute it $R_t(\mathcal{I}, \omega_l, \omega_u)$, to estimate the time cost of CANNIER+RERUN, $C_{\mathcal{T}}^{Rerun}(\mathcal{I}, \omega_l, \omega_u)$. It can find C_t from the output of `pytest-CANNIER` in the Features mode, since this is the third feature in Table 5.1. As for $R_t(\mathcal{I}, \omega_l, \omega_u)$, when $P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I}))$ is not in the ambiguous region between ω_l and ω_u , CANNIER+RERUN does not delegate to RERUN and so it never executes t ($R_t(\mathcal{I}, \omega_l, \omega_u) = 0$). Otherwise, when $y_{t,1} = 0$, RERUN would execute t exactly R_{max} times since t is not NOD flaky and therefore RERUN would never observe an inconsistent outcome ($R_t(\mathcal{I}, \omega_l, \omega_u) = R_{max}$). If $y_{t,1} = 1$, RERUN would execute t until it either observes an inconsistent outcome or reaches a limit of R_{max} runs. I refer to the final run number where either of these conditions are met as r_t . In this case, $R_t(\mathcal{I}, \omega_l, \omega_u)$ is the expected value of the discrete, finite distribution $P(r_t = x)$. The probability of t giving an inconsistent outcome after exactly x runs is $Exact(t, x)$. This is the probability of t failing $x - 1$ times and then passing once, or passing $x - 1$ times and then failing once. When $x < R_{max}$, $P(r_t = x) = Exact(t, x)$. However, when $x = R_{max}$, $P(r_t = x)$ is the probability of t giving an inconsistent outcome after exactly R_{max} runs, $Exact(t, R_{max})$, or not giving an inconsistent outcome after reaching the limit of R_{max} runs. Where $\mathbb{E}[P]$ is the expected value of the distribution P , the definition of the time cost of CANNIER+RERUN, denoted $C_{\mathcal{T}}^{Rerun}$, is given by the following equation.

$$\begin{aligned}
C_t &= \frac{1}{N_F} \sum_{i=1}^{N_F} \mathbf{x}_{t,i,3}, \\
Exact(t, x) &= \left(\frac{B_t}{N_B}\right)^{x-1} \left(1 - \frac{B_t}{N_B}\right) + \left(1 - \frac{B_t}{N_B}\right)^{x-1} \frac{B_t}{N_B}, \\
P(r_t = x) &= \begin{cases} Exact(t, x) & \text{if } 1 < x < R_{max} \\ Exact(t, R_{max}) + (1 - \sum_{r=2}^{R_{max}} Exact(t, x)) & \text{if } x = R_{max} \end{cases}, \quad (5.6) \\
R_t(\mathcal{I}, \omega_l, \omega_u) &= \begin{cases} 0 & \text{if } P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_l \vee P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ R_{max} & \text{if } \omega_l \leq P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \wedge y_{t,1} = 0 \\ \mathbb{E}[P(r_t = x)] & \text{if } \omega_l \leq P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \wedge y_{t,1} = 1 \end{cases}, \\
C_{\mathcal{T}}^{Rerun}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}} C_t R_t(\mathcal{I}, \omega_l, \omega_u)
\end{aligned}$$

To estimate the time cost of CANNIER+IDFCCLASS, $C_{\mathcal{T}}^{iDFClass}(\mathcal{I}, \omega_l, \omega_u)$, for a given test suite \mathcal{T} , CANNIER-Framework requires the number of times that IDFCCLASS is expected to attempt to classify each test case $t \in \mathcal{T}_2$ as either NOD or a victim, $\Gamma_t(\mathcal{I}, \omega_l, \omega_u)$. As before, when $P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I}, \omega_l, \omega_u))$ is not in the ambiguous region, CANNIER+IDFCCLASS does not delegate to IDFCCLASS and so it never classifies t ($\Gamma_t(\mathcal{I}, \omega_l, \omega_u) = 0$). Otherwise, IDFCCLASS will classify a test case after its first failure during the Classification stage and will reclassify a percentage of the additional failures as determined by γ . I assume that any test case undergoing classification by IDFCCLASS has a uniform probability of appearing at any position in the original and modified test run orders. Under this assumption, the mean length of the truncated original and modified

orders would both be equal to half the size of the test suite. Therefore, the mean time cost of classifying a single test case is equal to that of one full test suite run, as given by the following equation.

$$\Gamma_t(\mathcal{I}, \omega_l, \omega_u) = \begin{cases} 0 & \text{if } P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_l \vee P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ 1 + \gamma(S_t - 1) & \text{if } \omega_l \leq P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \end{cases}, \quad (5.7)$$

$$C_{\mathcal{T}}^{iDFClass}(\mathcal{I}, \omega_l, \omega_u) = \left(\sum_{t \in \mathcal{T}_2} \Gamma_t(\mathcal{I}, \omega_l, \omega_u) \right) \sum_{t \in \mathcal{T}} C_t$$

CANNIER-Framework estimates the detection performance of **CANNIER+PAIRWISE** as the ratio of victim-polluter pairs that would be detected by **PAIRWISE** to all such pairs in a given test suite \mathcal{T} . I selected this simpler metric, as opposed to MCC, because I assume that **CANNIER+PAIRWISE** will never incorrectly label a pair of test cases as having a victim-polluter relationship when they do not (false-positive). Under this assumption, this metric is equivalent to *true-positive rate* (TPR), also known as *sensitivity*. It has a range between 0 and 1, where 0 indicates that **CANNIER+PAIRWISE** detected none of the victim-polluter pairs and 1 indicates that it detected all of them. Since I designed the framework to only consider non-NOD flaky tests as candidate victims, such that they all have a reliable expected outcome, I have sufficient assurance that the assumption holds. Recall from Section 5.3.3 that **CANNIER+PAIRWISE** builds a set of victims $\mathcal{T}_V(\mathcal{I}, \omega_V)$, and polluters $\mathcal{T}_P(\mathcal{I}, \omega_P)$, given victim- and polluter-thresholds ω_V and ω_P . It then executes **PAIRWISE** with only the pairs in $\mathcal{T}_P(\mathcal{I}, \omega_P) \times \mathcal{T}_V(\mathcal{I}, \omega_V)$. **CANNIER-Framework** builds these sets using the predicted probabilities from the Victim-vs-Rest ($\phi = 3$) and Polluter-vs-Rest ($\phi = 4$) problems. The framework calculates TPR by dividing the number of victim-polluter pairs in $\mathcal{T}_P(\mathcal{I}, \omega_P) \times \mathcal{T}_V(\mathcal{I}, \omega_V)$ by the number of such pairs in $\mathcal{T} \times \mathcal{T}$. In other words, it divides the number of true-positives (TP) by the number of positives (P). To know how many pairs are in both sets, **CANNIER-Framework** relies on the outcomes recorded by **pytest-CANNIER** in the Victim mode. The framework estimates the time cost of **CANNIER+PAIRWISE**, $C_{\mathcal{T}}^{Pairwise}(\mathcal{I}, \omega_V, \omega_P)$, based on the sizes of both sets and the individual time costs of their members. The definition of $TPR_{\mathcal{T}}$ and the time cost of **CANNIER+PAIRWISE**, denoted $C_{\mathcal{T}}^{Pairwise}$, is provided by the following equation.

$$\begin{aligned} \mathcal{T}_V(\mathcal{I}, \omega_V) &= \{v | v \in \mathcal{T}, P(y_{t,3} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_V\}, \\ \mathcal{T}_P(\mathcal{I}, \omega_P) &= \{p | p \in \mathcal{T}, P(y_{t,4} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_P\}, \\ TP_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P) &= \sum_{p \in \mathcal{T}_P(\mathcal{I}, \omega_P)} |\{v | v \in \mathcal{T}_V(\mathcal{I}, \omega_V) - \{p\}, o_{p,v} \neq o_v\}|, \\ P_{\mathcal{T}} &= \sum_{p \in \mathcal{T}} |\{v | v \in \mathcal{T} - \{p\}, o_{p,v} \neq o_v\}|, \\ TPR_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P) &= \frac{TP_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P)}{P_{\mathcal{T}}}, \\ C_{\mathcal{T}}^{Pairwise}(\mathcal{I}, \omega_V, \omega_P) &= \left(|\mathcal{T}_P(\mathcal{I}, \omega_P)| \sum_{v \in \mathcal{T}_V(\mathcal{I}, \omega_V)} C_v \right) + \left(|\mathcal{T}_V(\mathcal{I}, \omega_V)| \sum_{p \in \mathcal{T}_P(\mathcal{I}, \omega_P)} C_p \right) \end{aligned} \quad (5.8)$$

5.5 Empirical Evaluation

I conducted experiments to answer the following research questions:

RQ1. How effective is machine learning-based flaky test detection?

RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?

RQ4. What impact does **CANNIER** have on the performance and time cost of rerunning-based flaky test detection?

5.5.1 Subject Set

For this study’s subject set, I used the test suites of the 26 open-source Python projects² studied in Chapter 4. I selected these at random from a list of projects critical to open-source infrastructure created by the Open Source Security Foundation of [260]. For this study, I randomly selected four more projects to improve the generalisability of the results. I used **CANNIER-Framework** to produce a dataset from these 30 test suites that contains 89,668 tests. I set the framework to perform 2,500 runs of each test suite in the Baseline mode of **pytest-CANNIER** ($N_B = 2500$), 2,500 runs in the Shuffle mode ($N_S = 2500$), and 30 runs in the Features mode ($N_F = 30$). Table 5.3 shows each project’s GitHub repository; the total number of tests ($|\mathcal{T}|$); the number of NOD flaky tests ($|\mathcal{T}_1^+|$), victims ($|\mathcal{T}_3^+|$), and polluters ($|\mathcal{T}_4^+|$); the number of victim-polluter pairs ($\sum_{p \in \mathcal{T}} |\{v | v \in \mathcal{T} - \{p\}, o_{p,v} \neq o_v\}|$); and the combined mean time cost of every test in seconds ($\sum_{t \in \mathcal{T}} [\frac{1}{N_F} \sum_{i=1}^{N_F} \mathbf{x}_{t,i,3}]$).

The projects of the subject set cover a wide variety of topics. All are hosted on the Python Package Index [265] that allows developers to associate them with zero or more “topic classifiers”. Topic classifiers are multi-level, for example: *Software Development :: Libraries :: Python Modules*. A developer may also specify a parent classifier on its own (e.g., just *Software Development*). Table 5.4 lists the topic classifiers of the 30 Python subjects. It also provides the frequencies of each classifier, taking into account their hierarchical nature.

5.5.2 Methodology

RQ1. How effective is machine learning-based flaky test detection?

The motivation behind this question is to establish a baseline for the performance of machine learning models for detecting flaky tests. While several studies have addressed this question for NOD flaky tests [8, 128, 159], and I addressed it for victims in the Chapter 4, no previous study has addressed it for polluters. It is important to consider polluters when answering RQ1 since they offer developers useful information when repairing victim flaky tests and are a necessary input to techniques for mitigating them [91, 119, 141].

I used **CANNIER-Framework** to evaluate 24 concrete machine learning pipelines for each of the four flaky test classification problems. I derived these from the combination of two choices of classifier type, four choices of classifier configuration, and three choices of data balancing technique. These choices form the concrete instantiations of the “Data Balancing” and “Model” blocks in the general pipeline from Figure 5.3. The two classifier types I considered were *Random Forest* [18, 143] and *Extra Trees* [51]. I selected these due to their success in Chapter 4 and the related work of other authors [8]. These are ensemble classifiers that can predict the probability that a test case is flaky based on a set of *Decision Tree* classifiers [137].

A *Decision Tree* is essentially a collection of nested if-then-else decision rules concerning the feature values. A given input traces a path through the *Decision Tree*, and the predicted probability of that input being of a given class is based on the class proportions of the training inputs with the same trace. When training a *Random Forest* classifier, each *Decision Tree* is fitted using a bootstrap sample (i.e., with replacement) of the training set and a random subset of features. The training procedure for each *Decision Tree* finds the decision rules that best discriminate between the positive and negative class (i.e., flaky or not flaky). When training an *Extra Trees* classifier, each *Decision Tree* is constructed using the entire training set, but still with random subsets of features. Furthermore, the training procedure selects the decision rules at random. For both *Random Forest* and *Extra Trees*, the final predicted probability of flakiness for a given test case is the average of the predicted probabilities of each *Decision Tree*.

The choices of classifier configuration were four values for the number of decision trees used by the *Random Forest* or *Extra Trees* classifier. These values were 25, 50, 75, and 100. In Chapter 4, I only considered *Random Forest* and *Extra Trees* models with 100 decision trees — the default value of my selected implementation [271]. Finally, for the three choices of data balancing, I

²I only reused the projects themselves as subjects. I did not reuse any of the data from the previous study.

Table 5.3: The 30 open-source Python projects examined in this study. The **Tests** column is the total number of test cases. The following three indicate the number of **NOD** flaky tests, **Victims**, and **Polluters**. The **Pairs** column gives the number of victim-polluter pairs. The **Cost** column is the combined mean time cost of every test case in seconds. The final row gives the totals for the whole subject set. Since `pytest-CANNIER` identifies polluters the same way as `PAIRWISE`, only considering pairs of test cases, polluters involved in more complex order-dependencies are not included. This is why the table shows that some projects appear to have victims without polluters.

GitHub Repository	Tests	NOD	Victims	Polluters	Pairs	Cost (s)
apache/airflow	3251	66	279	3241	45819	7.77×10^2
celery/celery	2332	-	15	17	24	1.31×10^2
quantumlib/Cirq	12048	-	17	2	32	8.67×10^2
conan-io/conan	3687	-	13	13	18	1.48×10^3
dask/dask	8015	1	1	37	37	1.34×10^3
encode/django-rest-framework	1402	-	1	3	3	2.63×10^3
spesmilo/electrum	542	1	1	2	2	5.99×10^1
Flexget/Flexget	1330	1	4	3	4	1.73×10^3
fonttools/fonttools	3448	1	42	-	-	1.19×10^2
graphql-python/graphql	346	-	1	1	1	1.73×10^1
facebookresearch/hydra	1538	-	19	348	952	1.77×10^2
HypothesisWorks/hypothesis	4348	5	6	3699	7401	3.92×10^3
ipython/ipython	807	6	297	796	118869	1.10×10^2
celery/kombu	1024	2	23	20	63	3.62×10^1
apache/libcloud	9809	3	133	471	1686	2.66×10^2
Delgan/loguru	1255	4	21	6	26	6.23×10^1
mitmproxy/mitmproxy	1232	-	18	338	735	3.12×10^1
python-pillow/Pillow	2567	-	26	2	26	9.38×10^1
PrefectHQ/prefect	7035	25	20	227	230	1.56×10^3
PyGithub/PyGithub	711	-	4	678	2712	5.55×10^1
Pylons/pyramid	2633	-	4	252	383	5.98×10^1
psf/requests	535	5	-	-	-	1.40×10^2
saltstack/salt	2672	12	4	65	65	2.52×10^2
scikit-image/scikit-image	6275	-	12	5882	5890	2.54×10^3
mwaskom/seaborn	1020	-	8	1	7	5.01×10^2
pypa/setuptools	694	1	23	4	4	2.08×10^2
sunpy/sunpy	1857	-	2	9	9	4.31×10^2
tornadoweb/tornado	1159	1	1	-	-	4.03×10^1
urllib3/urllib3	1320	15	1	-	-	8.57×10^1
xonsh/xonsh	4776	9	19	3114	9459	1.81×10^2
Overall	89668	158	1015	19231	194457	1.99×10^4

evaluated the *synthetic minority oversampling technique* (SMOTE) [23], SMOTE combined with *edited nearest-neighbors* (SMOTE+ENN), and SMOTE with *Tomek links* [153] (SMOTE+Tomek). SMOTE performs *oversampling*, meaning it produces synthetic data points of the minority class via interpolation. The ENN and Tomek techniques on their own perform *undersampling*, meaning they remove data points of the majority class based on similarity with their neighbors. The combination of these with SMOTE produces a hybrid balancing approach.

For each of the $24 \times 4 = 96$ concrete machine learning pipelines, I fixed the feature sample size at a single sample ($n_F = 1$) and had the framework repeat the classifier training and evaluation procedure 30 times (see Figure 5.3), using a different random sample \mathcal{I} to produce the mean feature vectors every time. In each instance, this resulted in 30 values of $P(y_{t,\phi} = 1)$ for every test case t and problem ϕ . To evaluate the performance of the pipelines, **CANNIER-Framework** needed predicted labels to calculate the confusion matrix category frequencies and MCC against the ground-truth labels for each problem. To produce the predicted labels to address this research question, I substituted $z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)$ in Equation 5.5 for the following definition of $z_{t,\phi}(\mathcal{I})$ that assigns a test case to its most likely class:

$$z_{t,\phi}(\mathcal{I}) = \begin{cases} 0 & \text{if } P(y_{t,\phi} = 1 | \mathbf{X}_t(\mathcal{I})) < 0.5 \\ 1 & \text{if } P(y_{t,\phi} = 1 | \mathbf{X}_t(\mathcal{I})) \geq 0.5 \end{cases} \quad (5.9)$$

With these predicted labels, I used **CANNIER-Framework** to calculate the confusion matrix category frequencies and the MCC of the 96 pipelines with respect to each of the 30 subject test suites in turn. I also had the framework calculate this with respect to the whole subject set for each pipeline by summing the category frequencies for each project and calculating the overall MCC from this total. This is to provide an individual assessment with respect to each test suite as well as an overview for the whole subject set. For the per-project and overall evaluations, **CANNIER-Framework** calculated mean values for the category frequencies and the MCC over the 30 repeats of classifier training and evaluation. This is to offer an evaluation that is more reliable given the non-determinism inherent to the machine learning models, the data balancing techniques, and potentially the dynamic feature values.

RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

In previous studies on machine learning-based flaky test detection with dynamic test case features, researchers performed only a single instrumented test suite run to create the feature vectors. The rationale for this question is to investigate the impact of using feature vectors that are the mean from multiple instrumented test suite runs. In the context of this study, that is multiple runs in the Features mode of **pytest-CANNIER**. This is to mitigate against the possible variance in the dynamic features. As an example, previous studies have found that the line coverage of test cases can vary across repeated executions [70, 139, 160]. Since three features in Table 5.1 are based on line coverage, I expect there to be some degree of noise in their values for each test case that could impact the detection performance of the classifier.

I took the best machine learning pipeline (in terms of the overall MCC) for each classification problem from the previous research question and followed the same methodology for training and evaluation, except I gave **CANNIER-Framework** a range of values for n_F to produce \mathcal{I} between 1 and 15 samples inclusive. With 30 repeats of classifier training and evaluation for each value of n_F , this resulted in $15 \times 30 = 450$ rounds of stratified 10-fold cross validation for each problem. This process enabled us to investigate the correlation between the number of repeated measurements to produce the mean feature vectors and the MCC of the resultant classifier.

RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?

In the interest of model explainability, I set out to investigate the impact of each individual feature in Table 5.1. To address this question, I applied the *Shapely Additive Explanations* (SHAP) technique [103]. It leverages concepts from game theory to quantify the contribution of an individual

Table 5.4: The topic classifiers of the subject projects and their frequencies. If a project declares derived classifiers I also incremented the frequencies of the parent classifiers. For a example, if a project declares *Internet*, *Internet :: Proxy Servers*, and *Software Development :: Libraries*, I would increment the frequencies of *Internet*, *Internet :: Proxy Servers*, *Software Development*, and *Software Development :: Libraries*.

Topic Classifier	Frequency
Communications	1
Education	1
Education :: Testing	1
Internet	5
Internet :: Proxy Servers	1
Internet :: WWW/HTTP	5
Internet :: WWW/HTTP :: WSGI	1
Multimedia	3
Multimedia :: Graphics	3
Multimedia :: Graphics :: Capture	1
Multimedia :: Graphics :: Capture :: Digital Camera	1
Multimedia :: Graphics :: Capture :: Screen Capture	1
Multimedia :: Graphics :: Graphics Conversion	2
Multimedia :: Graphics :: Viewers	1
Scientific/Engineering	4
Scientific/Engineering :: Physics	1
Scientific/Engineering :: Visualization	1
Security	1
Software Development	12
Software Development :: Build Tools	1
Software Development :: Libraries	7
Software Development :: Libraries :: Python Modules	3
Software Development :: Object Brokering	1
Software Development :: Testing	2
System	9
System :: Archiving	1
System :: Archiving :: Packaging	1
System :: Clustering	1
System :: Distributed Computing	4
System :: Logging	1
System :: Monitoring	1
System :: Networking	2
System :: Networking :: Monitoring	1
System :: Shells	1
Text Processing	1
Text Processing :: Fonts	1
Utilities	1

feature to the output value of a machine learning model for an individual data point. As inputs, SHAP takes a feature matrix and a model and returns a matrix of *SHAP values* in the same shape as the feature matrix. The SHAP value at (i, j) in the matrix represents the contribution of the j th feature on the model output for the i th data point relative to the mean output value over the dataset. This is such that summing the rows of the SHAP value matrix and adding the mean output value gives the original model output values.

In the context of this study, the features are those in Table 5.1, the data points are test cases, and the model output values are the predicted probabilities of each test case being in the positive class for a given flaky test classification problem. As the feature matrix, I used the mean feature vector for each test case over the 30 runs of `pytest-CANNIER` in the Features mode ($n_F = N_F$). As the machine learning model, I used `CANNIER-Framework` to train the best pipeline from RQ1 using the mean feature matrix. I did this for each of the four classification problems.

Once I had a SHAP value matrix for each problem, I ranked every feature in terms of their mean absolute SHAP value over every test case. A high value would indicate that the feature has a significant impact on the classifier’s decision (regardless of whether the impact is in favour of the negative class or the positive) and a low value would suggest the opposite. I then retrained the best pipeline for each problem with just the top 15, 12, 9, 6, and 3 features (with 30 repeats in each case). This is to observe the effect of dropping the less impactful features on the performance of the classifier.

RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

The motivation behind this research question is to investigate if CANNIER is able to reduce the time cost of rerunning-based flaky test detection techniques while maintaining good detection performance. For the application of CANNIER to the three techniques from Section 5.2, I used `CANNIER-Framework` to calculate the detection performance and single-core time cost associated with every point in a sample of their parameter spaces. For `CANNIER+RERUN` and `CANNIER+IDFCCLASS`, the space represents the values of the 3-tuple $(\omega_l, \omega_u, n_F)$, that is, the lower-threshold, the upper-threshold, and the number of samples to produce the mean feature vectors. In the case of `CANNIER+RERUN`, since R_{max} (the maximum number of times to execute a test case without observing an inconsistent outcome) is a parameter of the underlying `RERUN` technique, rather than a parameter introduced by CANNIER, I kept its value fixed at N_B (the number of test suite runs in the Baseline mode: 2,500). Similarly, for `CANNIER+IDFCCLASS`, I fixed the value of γ (the percentage of additional failures to recheck) to 20% because it is a parameter of `IDFCCLASS` and not one introduced by CANNIER. This particular value was recommended by the authors of `IDFLAKIES` [90]. For the detection performance and time cost of a given point for `CANNIER+RERUN/CANNIER+IDFCCLASS`, `CANNIER-Framework` calculated the mean over the 30 sets of predicted probabilities for the NOD-vs-Rest/NOD-vs-Victim problem from the 30 repeats of classifier training and evaluation for the given value of n_F from RQ2. For `CANNIER+PAIRWISE`, the parameter space represents $(\omega_V, \omega_P, n_F)$, the victim-threshold, the polluter-threshold, and the number of samples once more. In this case, the framework calculated the mean detection performance and time cost over 30 random pairs of the 30 sets of predicted probabilities for the Victim-vs-Rest problem and the 30 sets for the Polluter-vs-Rest problem for the given value of n_F .

For the sample of points in $(\omega_l, \omega_u, n_F)$, I used the values for ω_l from 0 to 1 inclusive with a step of 0.01, the values for ω_u from ω_l to 1.01 inclusive with a step of 0.01, and the values for n_F in the closed integer range from 1 to 15, except when $\omega_l = 0 \wedge \omega_u = 1.01$, in which case $n_F = 0$. The reason for starting from ω_l and going up to 1.01 for ω_u is to ensure that $\omega_l \leq \omega_u$ always holds and so that `CANNIER-Framework` evaluates the points where there is no upper-threshold on $P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I}))$ (see the second clause of Equation 5.4). The reason that $n_F = 0$ when $\omega_l = 0 \wedge \omega_u = 1.01$ is to indicate that the machine learning classifier, and therefore feature collection, is redundant because the ambiguous region is the entire range of $P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I}))$ under these conditions. Therefore, `CANNIER+RERUN` and `CANNIER+IDFCCLASS` reduce to

the original rerunning-based RERUN and IDFCCLASS respectively (see the third clause of Equation 5.4). As the sample of points in $(\omega_V, \omega_P, n_F)$, I used the values for both ω_V and ω_P from 0 to 1 inclusive with a step of 0.01. This excludes 1.01, since when one or both thresholds is greater than 1, the set of victims and/or polluters is empty and therefore PAIRWISE has nothing to do since $\mathcal{T}_V(\mathcal{I}, \omega_V) \times \mathcal{T}_P(\mathcal{I}, \omega_P) = \emptyset$. For n_F , the framework considers from 1 to 15, except when $\omega_V = \omega_P = 0$, where $n_F = 0$. The reason that $n_F = 0$ in this case is to indicate that the classifier is redundant because $\mathcal{T}_V(\mathcal{I}, \omega_V) = \mathcal{T}_P(\mathcal{I}, \omega_P) = \mathcal{T}$ and thus CANNIER+PAIRWISE reduces to original PAIRWISE.

I had CANNIER-Framework add the time taken to collect features to the overall time cost for each point. Since many features are dynamic, they require n_F test suite runs to measure, making the time cost of doing so $n_F \sum_{t \in \mathcal{T}} C_t$ for some test suite \mathcal{T} . For the points where $n_F = 0$, where the other parameters render the machine learning classifier redundant, this additional time cost is zero. I did not consider the time cost associated with applying the classifier to each test case because it is negligible relative to the time taken to execute the test suite (see Table 4.4). I also did not consider the time taken to train the classifier as part of the time cost of applying it. This is because the classifier only needs to be trained once and can then be applied any number of times, making training an off-line stage with a cost that can be amortised across uses.

I used CANNIER-Framework to compute the two-dimensional Pareto fronts of detection performance and time cost, with respect to the whole subject set, for the sample of points for CANNIER+RERUN, CANNIER+IDFCCLASS, and CANNIER+PAIRWISE. In this context, the Pareto front represents the subset of points such that, for each point, the detection performance is the greatest compared to all other points with the same time cost. To answer this research question, I compared the detection performance and time cost associated with the point representing the balanced application of CANNIER to the point where it reduces to the original rerunning-based detection technique, for each of the three fronts. As the point representing balanced CANNIER, I used the *knee point*. The knee point is the point with the smallest Euclidean distance to the *utopia point* on the Pareto front [170]. The utopia point represents a “perfect” solution that does not necessarily exist. In the context of this study, that would be the point with a detection performance of 1, for either MCC or true-positive rate (TPR), and a time cost of 0 seconds. For CANNIER+RERUN and CANNIER+IDFCCLASS, I also considered the point where they reduce to pure machine learning-based detection as an additional baseline. For this special case, I used the point on the Pareto front with the greatest MCC that also satisfies $\omega_l = \omega_u$. For all points that satisfy this condition, the techniques never defer to RERUN or IDFCCLASS because there is no ambiguous region between the two thresholds. For CANNIER+PAIRWISE, there is no such point, because it only limits the problem space for PAIRWISE but nevertheless always defers to it.

5.5.3 Threats to Validity

When deciding the ground-truth labels, CANNIER-Framework could incorrectly label some flaky tests as non-flaky. I used the framework to execute every test suite 2,500 times in their original test run orders to identify NOD flaky tests and 2,500 times in shuffled orders to identify victims. Given the non-deterministic nature of flaky tests, it is generally not possible to label a test case as non-flaky with complete certainty [68]. I mitigated this issue by having CANNIER-Framework perform as many reruns as possible within the limits of the available computational resources. In total, this stage required over six weeks of computational time on a computer with a 24-core AMD Ryzen 5900X CPU. While confidence in the label increases with the number of reruns, so too does the computational cost. In Chapter 4, I found the relationship between the number of detected flaky tests and the number of test suite reruns to be sublinear (see Figure 4.1). This finding supports another previous study, the authors of which identified a similar relationship [8]. This implies that continuing to re-execute a test suite gives diminishing returns with respect to the confidence of labelling a test case as non-flaky. This encourages us that the overall results of this study would be the same had the plugin performed more reruns, because it is unlikely that it would have detected significantly more flaky tests. Furthermore, `pytest-CANNIER` is unlikely to

detect certain flaky test categories by rerunning alone. For example “implementation-dependent” flaky tests may require changes to standard library implementations to manifest [140, 174]. The only category I made specific arrangements to detect were victims and their polluters; other special categories are out of the scope of this study.

The concrete machine learning pipelines of the Random Forest/Extra Trees classifier with SMOTE data balancing and the 18 features in Table 5.1 may unfairly represent machine learning-based flaky test detection. A whole host of previous studies [8, 19, 20, 65, 128, 130] identified Random Forest to be the most suitable type of machine learning classifier for detecting flaky tests. In the Chapter 4, I found that the Extra Trees classifier, a variant of Random Forest, was better suited for detecting flaky tests in some cases. Furthermore, the 18 features are based on the 16 features of FLAKE16 that I found to yield better detection performance when used to encode test cases compared to the previous state-of-the-art feature set [8]. This implies that my choice of pipeline and features is among the most suitable for detecting flaky tests currently in the literature.

There is a chance that **CANNIER-Framework** and **pytest-CANNIER** contain bugs that may go on to influence the results of the evaluation. Naturally, it is impossible to be totally sure that any non-trivial software system is totally free of bugs. However, I made sure to use well-established Python libraries for the bulk of the framework’s important functionality. These included **Coverage.py** [211] to measure line coverage, **psutil** [264] to measure many other dynamic test case properties, **Radon** [279] to measure source code metrics, **scikit-learn** [271] for an implementation of the Random Forest and Extra Trees classifier, and **shap** [281] to calculate the SHAP value matrices for RQ3. These are all popular open-source projects with many contributors, giving us confidence that any bugs would be identified, documented, and patched in a timely manner. I also wrote unit tests for greater confidence in the bespoke elements of **CANNIER-Framework** and **pytest-CANNIER**.

It is possible that the results of the study would not generalise to other Python projects outside of the 30 that I sampled, or to projects written in other programming languages. I randomly sampled 30 Python projects from a list of the top-200 most critical to open-source infrastructure, as determined by the Open Source Security Foundation [260]. Part of their metric for determining the criticality of a project is based on how many other projects declare a dependency on it. Therefore, any issues caused by flaky tests in these projects could potentially impact a wider portion of the Python ecosystem. Of course, this does not guarantee that the sample generalises to all Python projects, but does give us some assurance that the flaky tests I examined could represent a more serious problem compared to flaky tests in less critical projects. Without extending the subject set to include projects written in other languages, I cannot make any assurances that the results generalise outside of Python. Broadly speaking, however, the approach is language-agnostic. Considering Table 5.1, the 18 features could apply to almost any commonly used programming language. Therefore, I see no compelling reason to suggest that the results could not be reproduced with projects written in other languages, such as Java. In addition, it is possible that individual projects in the subject set with significantly more test cases than others could bias the overall results. For example, **airflow** had the highest number of NOD flaky tests at 66 — 264% of the second highest. To resolve this concern, **CANNIER-Framework** calculated performance metrics with respect to each individual project.

Given the empirical nature of this study, it may be difficult to reproduce the results. I took steps to make the methodology as repeatable as possible. Firstly, I included all scripts and software that I developed to facilitate this study in the replication package [217]. This includes the Dockerfile and requirements files for generating Python virtual environments [195]. Secondly, any aspects of the study that could be impacted by non-determinism, such as producing the predicted probabilities of test cases being flaky, I repeated 30 times. As such, the final results reported in this study involve taking the mean across these 30 repeats. Finally, where any aspects of **CANNIER-Framework** relied on random number generators (such as when instantiating machine learning models), I made sure to set the seed to a constant value to ensure that the results are the same across repeated runs.

Table 5.5: The top-12 pipelines (out of 24) for each flaky test classification problem in terms of overall MCC. The MCC values are the mean over 30 repeats of classifier training and evaluation, rounded to three significant figures.

(a) NOD-vs-Rest				(b) NOD-vs-Victim			
Classifier	Trees	Balancing	MCC	Classifier	Trees	Balancing	MCC
Extra Trees	100	SMOTE	0.532	Random Forest	75	SMOTE	0.693
Extra Trees	100	+Tomek	0.529	Random Forest	100	SMOTE	0.690
Extra Trees	75	SMOTE	0.527	Extra Trees	100	SMOTE	0.689
Extra Trees	50	SMOTE	0.526	Extra Trees	50	SMOTE	0.686
Extra Trees	25	SMOTE	0.521	Extra Trees	75	SMOTE	0.686
Extra Trees	50	+Tomek	0.519	Random Forest	50	SMOTE	0.685
Extra Trees	75	+Tomek	0.519	Random Forest	75	+Tomek	0.684
Extra Trees	25	+Tomek	0.507	Random Forest	25	SMOTE	0.679
Random Forest	100	SMOTE	0.488	Random Forest	100	+Tomek	0.675
Random Forest	75	SMOTE	0.479	Extra Trees	100	+Tomek	0.673
Random Forest	50	SMOTE	0.479	Extra Trees	75	+Tomek	0.669
Random Forest	25	SMOTE	0.477	Random Forest	25	+Tomek	0.668

(c) Victim-vs-Rest				(d) Polluter-vs-Rest			
Classifier	Trees	Balancing	MCC	Classifier	Trees	Balancing	MCC
Extra Trees	75	SMOTE	0.520	Random Forest	100	SMOTE	0.946
Extra Trees	100	SMOTE	0.519	Random Forest	75	SMOTE	0.945
Extra Trees	50	SMOTE	0.518	Random Forest	50	SMOTE	0.944
Extra Trees	100	+Tomek	0.515	Random Forest	25	SMOTE	0.943
Extra Trees	75	+Tomek	0.513	Random Forest	100	+Tomek	0.941
Extra Trees	50	+Tomek	0.511	Extra Trees	100	SMOTE	0.941
Extra Trees	25	SMOTE	0.510	Random Forest	75	+Tomek	0.940
Extra Trees	25	+Tomek	0.502	Extra Trees	75	SMOTE	0.940
Random Forest	50	SMOTE	0.501	Random Forest	50	+Tomek	0.940
Random Forest	75	SMOTE	0.498	Extra Trees	50	SMOTE	0.939
Random Forest	100	SMOTE	0.498	Random Forest	25	+Tomek	0.937
Random Forest	25	SMOTE	0.490	Extra Trees	100	+Tomek	0.937

5.6 Results

5.6.1 RQ1. How effective is machine learning-based flaky test detection?

Table 5.5 shows the top-12 concrete machine learning pipelines (out of 24) for each flaky test classification problem in terms of overall MCC. Recall from Section 5.5.2 that these MCC values are with respect to the entire subject set and are the mean over 30 repeats of classifier training and evaluation (see Equation 5.5). Extra trees appears to be the best classifier for the NOD-vs-Rest and Victim-vs-Rest problems, and pipelines using Extra Trees are consistently at the top of these tables. For NOD-vs-Victim and Polluter-vs-Rest, the most performant classifier appears to be Random Forest, though with less consistency. In terms of data balancing, the best pipelines for each problem used plain SMOTE. Unlike SMOTE+Tomek, SMOTE+ENN did not make it into the top-12 for any problem. In all cases, the negative gradient of detection performance going down the table is small, such that the difference in overall MCC between the best pipeline and the 12th best pipeline is not that significant.

Tables 5.6 and 5.7 show the per-project and overall confusion matrix category frequencies (TN, FN, FP, TP) and MCC of the best pipeline for each flaky test classification problem. Table 5.6a shows the performance for the NOD-vs-Rest problem. The table lists relatively few projects with

Table 5.6: The per-project and overall results of the best pipelines from Table 5.5 for the NOD-vs-Rest (a) and NOD-vs-Victim (b) problems. The tables give the confusion matrix category frequencies (i.e., column labels **TN**, **FN**, **FP**, **TP**), rounded to the nearest integer, and the Matthews correlation coefficient (i.e., column label **MCC**). Captions give the mean (μ) and standard deviation (σ) of the per-project MCC. Values are the mean over 30 repeats of model training and evaluation. Dashes indicate that the value is exactly zero. The “ \perp ” symbol indicates that the value is not defined, which was caused by a division by zero when a project does not have any test cases of certain categories.

(a) NOD-vs-Rest ($\mu = 0.52, \sigma = 0.29$)						(b) NOD-vs-Victim ($\mu = 0.55, \sigma = 0.22$)					
Project	TN	FN	FP	TP	MCC	Project	TN	FN	FP	TP	MCC
airflow	3159	26	26	40	0.59	airflow	250	11	25	45	0.66
celery	2332	-	-	-	\perp	celery	14	-	1	-	\perp
Cirq	12048	-	-	-	\perp	Cirq	17	-	-	-	\perp
conan	3687	-	0	-	\perp	conan	13	-	0	-	\perp
dask	8014	1	0	-	\perp	dask	1	-	-	-	\perp
django-rest-...	1402	-	-	-	\perp	django-rest-...	0	-	1	-	\perp
electrum	540	1	1	-	\perp	electrum	0	1	1	0	\perp
Flexget	1329	1	-	-	\perp	Flexget	3	1	1	-	\perp
fonttools	3447	1	-	-	\perp	fonttools	42	-	-	-	\perp
graphene	346	-	-	-	\perp	graphene	1	-	-	-	\perp
hydra	1538	-	-	-	\perp	hydra	19	-	-	-	\perp
hypothesis	4341	5	2	-	0.00	hypothesis	6	3	0	0	\perp
ipython	800	2	0	4	0.76	ipython	296	2	1	4	0.71
kombu	1022	2	-	-	\perp	kombu	23	1	-	-	\perp
libcloud	9806	3	-	-	\perp	libcloud	133	3	0	-	\perp
loguru	1250	2	1	2	\perp	loguru	20	1	1	2	0.55
mitmproxy	1232	-	-	-	\perp	mitmproxy	6	-	0	-	\perp
Pillow	2567	-	0	-	\perp	Pillow	26	-	0	-	\perp
prefect	7005	13	5	12	0.58	prefect	17	1	3	16	0.79
PyGithub	711	-	-	-	\perp	PyGithub	4	-	-	-	\perp
pyramid	2633	-	-	-	\perp	pyramid	4	-	-	-	\perp
requests	530	2	-	3	0.82	requests	-	0	-	4	\perp
salt	2660	4	0	8	0.82	salt	3	1	1	11	0.69
scikit-image	6275	-	0	-	\perp	scikit-image	12	-	-	-	\perp
seaborn	1020	-	-	-	\perp	seaborn	8	-	0	-	\perp
setuptools	693	1	-	-	\perp	setuptools	23	0	0	1	\perp
sunpy	1857	-	-	-	\perp	sunpy	2	-	-	-	\perp
tornado	1157	1	1	-	\perp	tornado	0	-	1	-	\perp
urllib3	1295	13	10	2	0.16	urllib3	-	3	1	12	0.12
xonsh	4763	5	4	4	0.45	xonsh	14	3	5	6	0.33
Overall	89460	83	50	75	0.53	Overall	957	32	42	99	0.69

Table 5.7: The per-project and overall results of the best pipelines from Table 5.5 for the Victim-vs-Rest (a) and Polluter-vs-Rest (b) problems. See Table 5.6 caption for more details.

(a) Victim-vs-Rest ($\mu = 0.51, \sigma = 0.24$)						(b) Polluter-vs-Rest ($\mu = 0.46, \sigma = 0.34$)					
Project	TN	FN	FP	TP	MCC	Project	TN	FN	FP	TP	MCC
airflow	2880	81	92	198	0.67	airflow	0	5	10	3236	0.01
celery	2316	9	1	6	0.59	celery	2311	15	4	2	0.20
Cirq	12030	3	1	14	0.88	Cirq	12032	1	14	1	0.12
conan	3666	8	8	5	0.38	conan	3621	6	53	7	0.25
dask	8013	1	1	-	⊥	dask	7947	0	31	37	0.73
django-rest-...	1400	1	1	-	⊥	django-rest-...	1397	3	2	-	⊥
electrum	539	1	2	-	⊥	electrum	525	2	15	-	0.01
Flexget	1325	3	1	1	⊥	Flexget	1327	3	0	-	⊥
fonttools	3395	6	11	36	0.82	fonttools	3443	-	5	-	⊥
graphene	345	1	0	-	⊥	graphene	344	1	1	-	⊥
hydra	1513	13	6	6	0.37	hydra	1186	13	4	335	0.97
hypothesis	4341	3	1	3	0.57	hypothesis	563	11	86	3688	0.91
ipython	377	206	133	91	0.05	ipython	7	165	4	631	0.13
kombu	1000	12	1	11	0.68	kombu	1002	15	2	5	0.42
libcloud	9612	86	64	47	0.38	libcloud	9315	195	23	276	0.73
loguru	1225	5	9	16	0.69	loguru	1249	6	0	0	⊥
mitmproxy	1213	13	1	5	0.50	mitmproxy	880	71	14	267	0.82
Pillow	2530	18	11	8	0.37	Pillow	2534	2	31	0	0.02
prefect	7014	16	1	4	0.40	prefect	6781	184	27	43	0.33
PyGithub	707	1	0	3	⊥	PyGithub	17	8	16	670	0.58
pyramid	2629	3	0	1	⊥	pyramid	2355	52	26	200	0.82
requests	535	-	-	-	⊥	requests	530	-	4	-	⊥
salt	2668	4	0	-	⊥	salt	2605	16	2	49	0.85
scikit-image	6261	5	2	7	0.69	scikit-image	331	191	62	5691	0.71
seaborn	1009	8	3	0	⊥	seaborn	990	1	29	0	0.01
setuptools	668	5	3	18	0.81	setuptools	686	2	4	2	0.39
sunpy	1855	2	-	-	⊥	sunpy	1835	6	13	2	0.21
tornado	1156	1	2	-	0.00	tornado	1156	-	3	-	⊥
urllib3	1318	1	1	-	⊥	urllib3	1310	-	10	-	⊥
xonsh	4753	14	4	5	0.36	xonsh	1608	102	54	3012	0.93
Overall	88292	529	361	486	0.52	Overall	69889	1077	548	18154	0.95

a defined value for MCC because many in the subject set contain zero or only very few NOD flaky tests (see Table 5.3). The overall MCC for this problem is 0.53 and the mean per-project MCC is close at 0.52. Recall that **CANNIER-Framework** calculated the overall MCC from the overall confusion matrix category frequencies, that are the sum of the per-project frequencies. An MCC of 1 indicates a perfect classifier and an MCC of 0 indicates a classifier no better than random guessing. Therefore, the detection performance of the best pipeline for this problem was fairly lacklustre. Furthermore, the standard deviation of the per-project MCC is relatively high at 0.29, suggesting that the performance of the pipeline is quite variable between projects. This is further evident from the wide range of MCC values among the different projects. Table 5.6b shows the results for the NOD-vs-Victim problem. Once again, the table contains relatively few projects with an MCC value for the same reason as before. At 0.69, the overall MCC for this problem is greater than that for NOD-vs-Rest. Also, the standard deviation of the per-project MCC is lower at 0.22. However, the mean of 0.55 is considerably lower than the overall MCC.

Table 5.7a gives the performance for the Victim-vs-Rest problem. At 0.52, the overall MCC is very close to the mean per-project MCC of 0.51 and is comparable to that of NOD-vs-Rest. Unlike the previous two problems, there are many more projects with a defined value for MCC, since most test suites in the subject set contained victim flaky tests. Finally, Table 5.7b gives the results for the Polluter-vs-Rest problem. While the overall MCC is very high at 0.95, the mean per-project MCC is much lower at 0.46 and the standard deviation is the greatest of all four problems at 0.34.

Conclusion for RQ1 The overall MCC of the best pipelines for the four classification problems ranges from 0.95 for Polluter-vs-Rest to 0.52 for Victim-vs-Rest. For NOD-vs-Rest and Victim-vs-Rest, the mean per-project MCC is close to the overall MCC. The standard deviation of the per-project MCC ranges from 0.22 to 0.34. These findings suggest that the performance of machine learning-based flaky test detection is lackluster and variable between projects, motivating the need for an alternative approach.

5.6.2 RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

Figure 5.5 shows the relationship between the sample size to produce the mean feature vectors (n_F) and the overall detection performance (MCC) of the best pipeline for each classification problem. Recall from Section 5.4.2 that **CANNIER-Framework** encoded test cases with feature vectors that were the mean of a random sample (\mathcal{I}) of the output from 30 test suite runs in the Features mode of **pytest-CANNIER**. Figure 5.5a shows the relationship for the NOD-vs-Rest problem. At 0.86, the Spearman's rank correlation coefficient (ρ) indicates that the relationship is positive. However, the gradient (a) of the line of best fit (in red) is small at just 0.0014. The MCC when $n_F = 15$ on the line of best fit is only 4% greater than the MCC when $n_F = 1$. For the NOD-vs-Victim problem, Figure 5.5b indicates that the relationship is weaker with a correlation coefficient of 0.71. In this case, the gradient is even smaller (0.0007), with just a 1% increase in MCC from $n_F = 15$ to $n_F = 1$.

Figure 5.5c shows the relationship for Victim-vs-Rest. The correlation coefficient of 1.00 indicates a very strong positive correlation, as is clear from the plot. The gradient of the line of best fit is comparable to NOD-vs-Rest (0.0019). In the case of the Polluter-vs-Rest problem, Figure 5.5d also shows a very strong positive relationship between n_F and MCC with a corresponding correlation coefficient of 1. However, the gradient is very small (0.0008).

Conclusion for RQ2 The relationship between the sample size to produce the mean feature vectors and the overall MCC of the best pipeline is positive but of variable strength across the four flaky test classification problems. For all problems, particularly NOD-vs-Victim and Polluter-vs-Rest, the gradient is small. These results indicate that using mean feature vectors has a small but positive impact on detection performance.

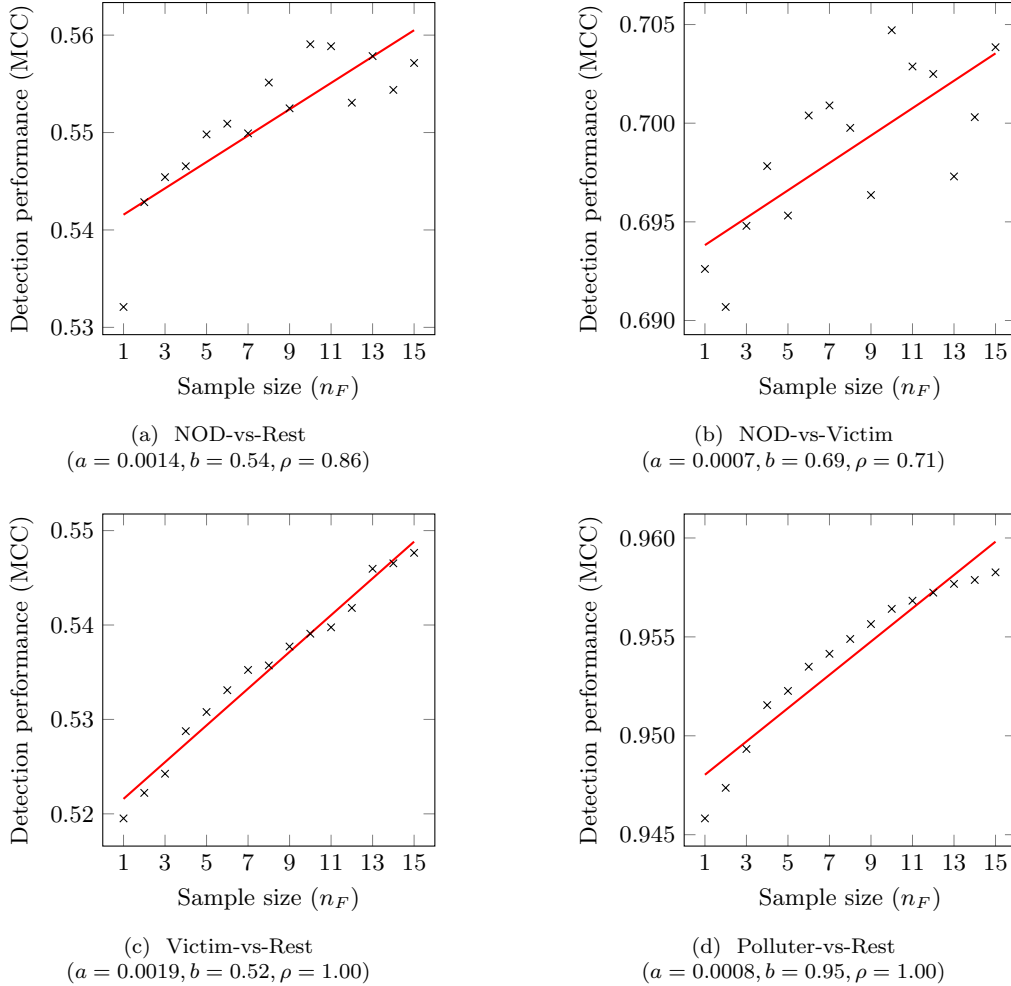


Figure 5.5: Plots showing that the relationship between the number of samples to produce the mean feature vectors (n_F) and the overall detection performance (MCC) of the best machine learning pipeline is positive but variable in terms of strength and gradient across the four problems. MCC values are the mean over 30 repeats. Captions give the coefficients of the red least-squares best-fit line ($MCC = a \times n_F + b$) and the Spearman's rank correlation coefficient (ρ).

5.6.3 RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?

Figure 5.6 shows the SHAP values for the four flaky test classification problems as beeswarm plots. In each plot, every feature in Table 5.1 is represented by a row, with each value in its corresponding column in the SHAP value matrix plotted as a coloured dot, for which there is one for every test case in the whole subject set. The horizontal position of each dot represents the SHAP value itself, with negative SHAP values towards the left and positive SHAP values towards the right, as indicated by the x-axis labels. Recall from Section 5.5.2 that a positive SHAP value means the contribution of the feature to the classifier output value from the best pipeline for a given test case and problem was positive (increased it). Conversely, a negative SHAP value means the contribution was negative (decreased it). In this context, the output value is the predicted probability of the test case belonging to the positive class of the problem. This means if a feature contributes positively to the output, it “pushes” the classifier towards predicting the positive class, and if it contributes negatively, it pushes towards the negative class. The colour of the dots represent the feature value relative to the mean feature value, with lower values coloured blue and higher values coloured red. For example, a blue dot on the left side of the x-axis indicates a test case with a relatively low feature value and a positive contribution. The vertical positions of the dots represent density, such that dots with similar SHAP values “swarm” around one another. From top-to-bottom, the features are in descending order of mean absolute SHAP value. In other words, the features closer to the top have a greater overall impact on the classifier output.

For the NOD-vs-Rest problem, the contribution of AST Depth, Run Time, Read Count, Context Switches, Write Count, Wait Time, Max. Children, and Test Lines of Code appears positive (towards predicting NOD flaky) when their values are high and negative when their values are low. This is evident from how the dots on the left side of their rows in Figure 5.6a are mostly blue and those on the right are mostly red. Conversely, the contribution of Assertions appears negative when high and positive when low, as visualised by mostly red dots on the left and mostly blue on the right. The contribution of some features appears more nuanced. For example, when the contribution of Covered Change is negative its value is mostly high. However, when its contribution is positive its value is mixed. For the NOD-vs-Victim problem (Figure 5.6b), Context Switches, Run Time, Max. Threads, Read Count, Write Count, Cyclomatic Complexity, External Modules, Max. Children, and Halstead Volume appear to contribute positively (towards predicting NOD flaky) when their values are high and negatively when low. The contribution of the individual features for this problem appear considerably less well-defined compared to NOD-vs-Rest. There are some similarities between the results for these two problems, such as Run Time, Read Count, Context Switches, Write Count, and Max. Children mostly contributing positively when high and negatively when low.

As shown by Figure 5.6c the contribution of Maintainability, Write Count, Read Count, and Wait Time features appear broadly positive when their values are high (towards predicting victim flaky) and negative when low. On the other hand, Source Covered Lines, Cyclomatic Complexity, and Halstead Volume show the opposite behaviour with moderate consistency. The impact of the features for this problem differs significantly compared to the NOD-vs-Victim problem. For example, the Maintainability and Cyclomatic Complexity feature appears to have nearly the exact opposite contribution pattern. Finally, for the Polluter-vs-Rest problem (Figure 5.6d), Run Time, Assertions, Halstead Volume, and Wait Time contribute positively when high. Covered Lines, Source Covered Lines, and Max. Children show the opposite contribution.

Figure 5.7 shows how the overall MCC of the best pipelines for each problem decreases as the number of features used by `CANNIER-Framework` to train the classifier are reduced, starting from the least impactful in terms of mean absolute SHAP value. For example, for the NOD-vs-Rest problem, the MCC value at 6 on the x-axis corresponds to a classifier that only considers AST Depth, Max. Threads, Run Time, Max. Memory, Read Count, and Context Switches. Initially, the detriment to detection performance is fairly small as only the least important features are pruned. At around 9 features, the overall MCC begins to fall for every problem.

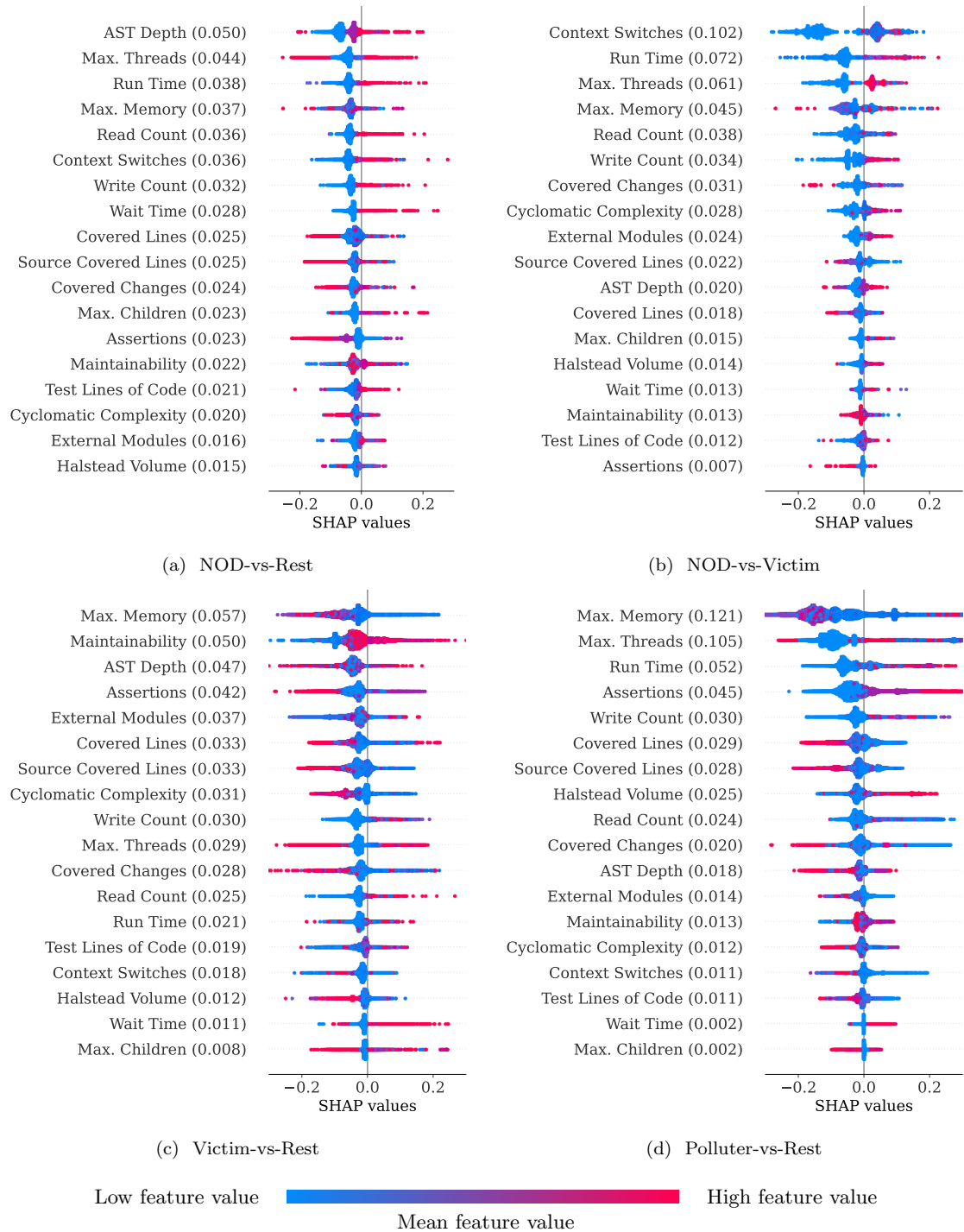


Figure 5.6: SHAP values for the four flaky test classification problems as beeswarm plots. These are based on the classifiers from best pipelines for each problem from RQ1. Blue dots represent lower feature values and red dots represent higher feature values. Purple dots represent feature values closer to the mean value. The vertical positions of the dots represent density, such that dots with similar SHAP values “swarm” around one another. Features are in descending order of their mean absolute SHAP value, which each beeswarm plot gives in parentheses. This is a measure of their overall impact on the classifier’s decision.

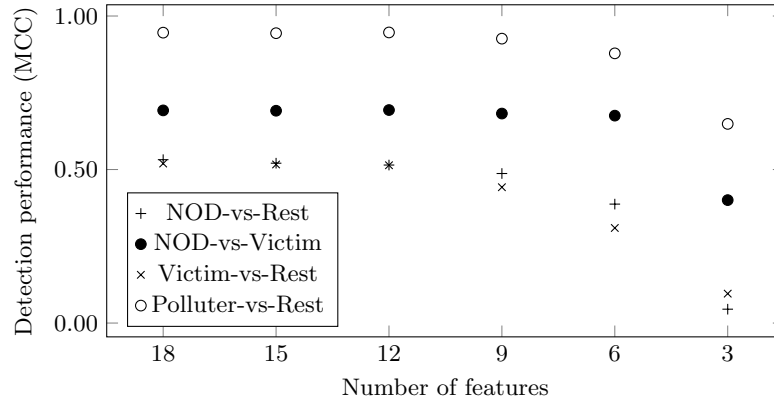


Figure 5.7: The relationship between the overall MCC of the best machine learning pipelines for each flaky test classification problem and the number of top features used by `CANNIER-Framework` to train the classifier in terms of mean absolute SHAP value. On the left side of the plot, only the less impactful features are removed, which has little effect on detection performance. Towards the right, the more impactful features are dropped, resulting in a significant reduction of MCC. MCC values are the mean over 30 repeats of classifier training and evaluation.

Conclusion for RQ3 In terms of mean absolute SHAP value, the most impactful features for the NOD-vs-Rest problem were AST Depth, Max. Threads, and Run Time; and for the NOD-vs-Victim problem were Context Switches, Run Time, and Max. Threads. For the Victim-vs-Rest problem, the most impactful were Max. Memory, Maintainability, and AST Depth; and for the Polluter-vs-Rest problem were Max. Memory, Max. Threads, and Run Time. Some features had a clear contribution pattern to the classifier and others less so, suggesting potentially more complex relationships.

5.6.4 RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

Figure 5.8 shows the Pareto fronts of overall detection performance and time cost for the application of CANNIER to the three rerunning-based detection techniques (see Equations 5.5, 5.6, 5.7, and 5.8). From right-to-left, the first pin on each curve is at the point representing the original rerunning-based technique (where the machine learning classifier becomes redundant). The second is at the point representing the balanced application of CANNIER (the knee point). Tables 5.8, 5.9, and 5.10 give the per-project and overall results at this point. For `CANNIER+RERUN` and `CANNIER+IDFCCLASS`, the third is at the point representing pure machine learning-based detection (greatest MCC where $\omega_l = \omega_u$). Above each pin in square brackets is the detection performance and time cost associated with the point (its coordinates on the axes). Below in parentheses are its parameters.

Figure 5.8a and Table 5.8a give the results for `CANNIER+RERUN`. As shown by the figure, the time cost associated with the point representing balanced `CANNIER+RERUN` (middle pin) is 89% lower than the time cost associated with the point representing original `RERUN` (right pin). At 0.92, the MCC at the balanced `CANNIER+RERUN` point is significantly greater than the MCC at the point representing pure machine learning-based detection (left pin), which is 0.55. As shown by the table, the per-project MCC is very consistent. Naturally, the MCC at the original `RERUN` point is exactly 1, since the predicted labels are the same as the ground-truth labels in this case (see Equation 5.4). Furthermore, the time cost at the pure machine learning point is significantly lower than the time cost at the other points of interest. This is because the only time cost associated with this point is that of collecting feature data. These results demonstrate that applying CANNIER to `RERUN` can significantly reduce its time cost while maintaining a

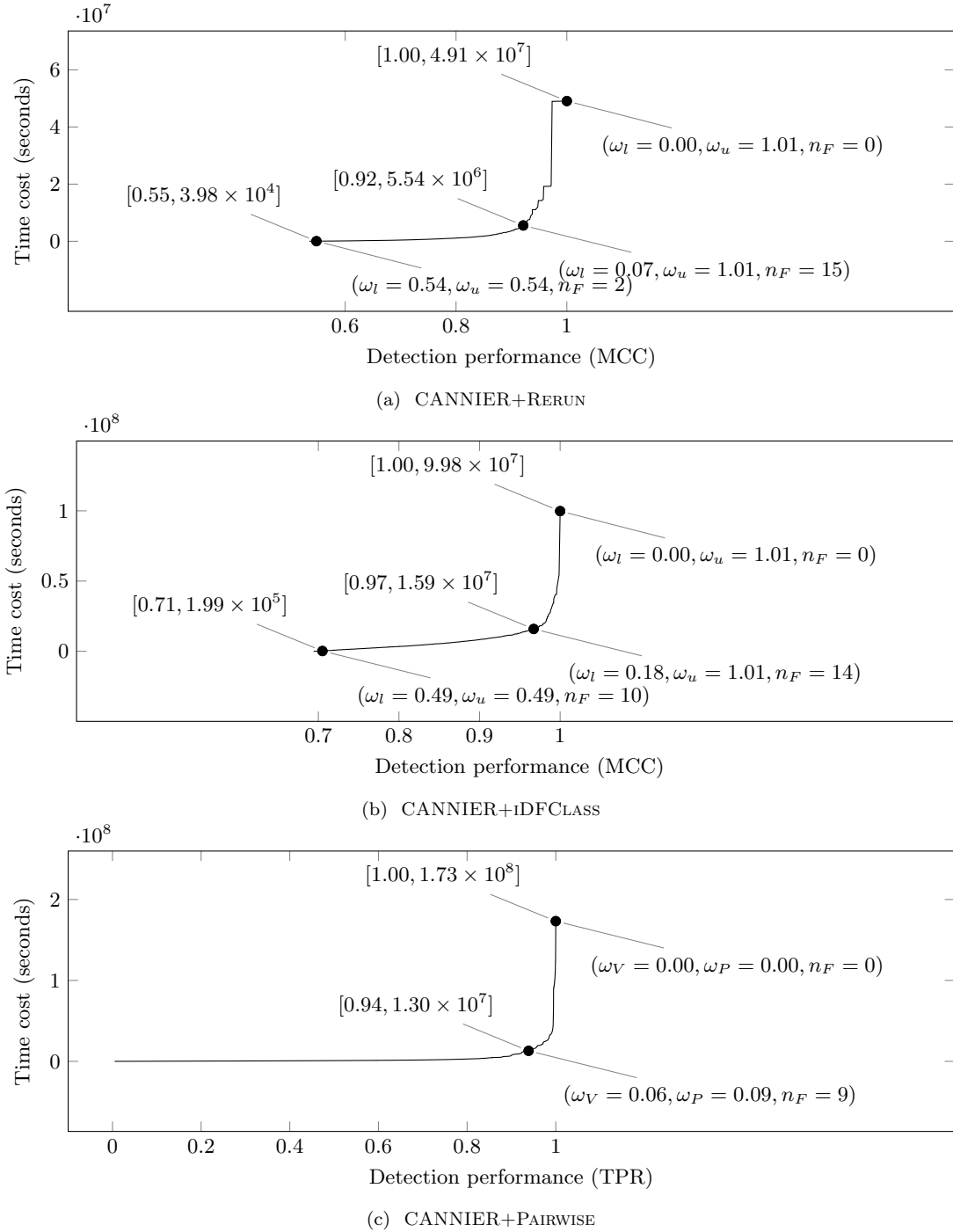


Figure 5.8: The Pareto fronts of detection performance and time cost for the application of CANNIER to the three rerunning-based detection techniques. From right-to-left, the first pin on each curve is at the point representing the original rerunning-based technique. The second is at the point representing the balanced application of CANNIER. For CANNIER+RERUN (a) and CANNIER+IDFCCLASS (b), the third is at the point representing pure machine learning-based detection. There is no third pin for CANNIER+PAIRWISE (c) because it is not possible to use a pure machine learning-based approach in this context (see Section 5.5.2). Above each pin in square brackets is the detection performance and time cost with respect to the whole subject set. Below in parentheses are the parameters.

Table 5.8: The per-project and overall results for CANNIER+RERUN. The table gives the confusion matrix categories, rounded to the nearest integer, and the MCC at the point in the parameter space representing balanced CANNIER+RERUN ($\omega_l = 0.07, \omega_u = 1.01, n_F = 15$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original RERUN (**Original**). The time cost is significantly reduced when using CANNIER. Values are the mean over 30 repeats of model training and evaluation. Dashes indicate that the value is exactly zero. The “ \perp ” symbol indicates that the value is not defined, which was caused by a division by zero when a project does not have any test cases of certain categories.

Project	TN	FN	FP	TP	MCC	Time cost (s)	
						CANNIER+	Original
airflow	3185	3	-	63	0.98	7.71×10^5	1.69×10^6
celery	2332	-	-	-	\perp	8.53×10^4	3.28×10^5
Cirq	12048	-	-	-	\perp	2.32×10^5	2.17×10^6
conan	3687	-	-	-	\perp	2.42×10^5	3.70×10^6
dask	8014	-	-	1	1.00	3.01×10^5	3.34×10^6
django-rest-...	1402	-	-	-	\perp	8.85×10^4	6.57×10^6
electrum	541	1	-	-	\perp	5.12×10^4	1.39×10^5
Flexget	1329	1	-	-	\perp	7.25×10^4	4.32×10^6
fonttools	3447	1	-	-	\perp	1.09×10^4	2.97×10^5
graphene	346	-	-	-	\perp	2.55×10^3	4.31×10^4
hydra	1538	-	-	-	\perp	2.34×10^4	4.42×10^5
hypothesis	4343	4	-	1	\perp	1.68×10^6	9.57×10^6
ipython	801	0	-	6	0.98	5.04×10^4	2.63×10^5
kombu	1022	1	-	1	\perp	1.27×10^4	9.05×10^4
libcloud	9806	1	-	2	\perp	5.00×10^3	6.66×10^5
loguru	1251	2	-	2	\perp	4.73×10^4	1.48×10^5
mitmproxy	1232	-	-	-	\perp	2.11×10^3	7.79×10^4
Pillow	2567	-	-	-	\perp	2.89×10^4	2.35×10^5
prefect	7010	2	-	23	0.96	4.05×10^5	3.80×10^6
PyGithub	711	-	-	-	\perp	1.41×10^4	1.39×10^5
pyramid	2633	-	-	-	\perp	1.10×10^3	1.49×10^5
requests	530	0	-	5	1.00	8.48×10^4	3.46×10^5
salt	2660	1	-	11	0.96	8.75×10^4	6.27×10^5
scikit-image	6275	-	-	-	\perp	5.73×10^5	6.36×10^6
seaborn	1020	-	-	-	\perp	6.41×10^4	1.25×10^6
setuptools	693	1	-	-	\perp	1.40×10^5	4.75×10^5
sunpy	1857	-	-	-	\perp	3.89×10^5	1.08×10^6
tornado	1158	1	-	-	\perp	6.44×10^3	1.01×10^5
urllib3	1305	3	-	12	0.89	3.53×10^4	2.13×10^5
xonsh	4767	1	-	8	0.94	2.94×10^4	4.50×10^5
Overall	89510	24	-	134	0.92	5.54×10^6	4.91×10^7

Table 5.9: The per-project and overall results for CANNIER+iDFCLASS. The table gives the confusion matrix categories, rounded to the nearest integer, and the MCC at the point in the parameter space representing balanced CANNIER+iDFCLASS ($\omega_l = 0.18, \omega_u = 1.01, n_F = 14$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original iDFCLASS (**Original**). See Table 5.8 caption for more details.

Project	TN	FN	FP	TP	MCC	Time cost (s)	
						CANNIER+	Original
airflow	275	3	-	53	0.97	9.22×10^6	7.24×10^7
celery	15	-	-	-	\perp	6.95×10^4	2.06×10^5
Cirq	17	-	-	-	\perp	3.23×10^4	4.47×10^6
conan	13	-	-	-	\perp	2.47×10^4	4.84×10^5
dask	1	-	-	-	\perp	1.87×10^4	6.60×10^5
django-rest-...	1	-	-	-	\perp	1.00×10^6	9.66×10^5
electrum	1	1	-	0	\perp	1.11×10^3	3.95×10^2
Flexget	4	1	-	0	\perp	1.44×10^6	1.75×10^6
fonttools	42	-	-	-	\perp	2.91×10^4	2.48×10^6
graphene	1	-	-	-	\perp	2.42×10^2	4.27×10^3
hydra	19	-	-	-	\perp	2.93×10^4	1.31×10^6
hypothesis	6	1	-	2	0.85	6.32×10^5	9.55×10^5
ipython	297	0	-	5	0.99	1.80×10^4	7.84×10^5
kombu	23	1	-	-	\perp	4.63×10^3	1.38×10^5
libcloud	133	1	-	2	\perp	8.73×10^4	8.74×10^6
loguru	21	0	-	3	0.98	4.15×10^4	2.35×10^5
mitmproxy	6	-	-	-	\perp	2.61×10^3	3.70×10^4
Pillow	26	-	-	-	\perp	2.55×10^3	1.51×10^4
prefect	20	-	-	17	1.00	7.84×10^5	1.04×10^6
PyGithub	4	-	-	-	\perp	7.82×10^2	3.11×10^2
pyramid	4	-	-	-	\perp	8.37×10^2	3.54×10^4
requests	-	-	-	4	\perp	1.81×10^4	1.61×10^4
salt	4	-	-	12	1.00	1.26×10^6	1.33×10^6
scikit-image	12	-	-	-	\perp	4.89×10^5	6.98×10^5
seaborn	8	-	-	-	\perp	7.64×10^4	1.29×10^5
setuptools	23	-	-	1	1.00	7.53×10^4	1.95×10^5
sunpy	2	-	-	-	\perp	6.20×10^3	1.94×10^5
tornado	1	-	-	-	\perp	6.05×10^2	4.03×10^1
urllib3	1	0	-	15	0.99	1.25×10^4	1.13×10^4
xonsh	19	1	-	8	0.95	5.18×10^5	5.75×10^5
Overall	999	8	-	123	0.97	1.59×10^7	9.98×10^7

Table 5.10: The per-project and overall results for CANNIER+PAIRWISE. The table gives the number of detected victim-polluter pairs (**TP**), the total number of such pairs (**P**), and the true-positive rate (**TPR**) at the point in the parameter space representing balanced CANNIER+PAIRWISE ($\omega_V = 0.06, \omega_P = 0.09, n_F = 9$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original PAIRWISE (**Original**). See Table 5.8’s caption for more details about the entities in this table.

Project	TP	P	TPR	Time cost (s)	
				CANNIER+	Original
airflow	45490	45819	0.99	2.60×10^6	5.05×10^6
celery	7	24	0.31	2.95×10^4	6.12×10^5
Cirq	30	32	0.94	1.15×10^5	2.09×10^7
conan	-	18	-	2.13×10^5	1.09×10^7
dask	1	37	0.03	2.39×10^5	2.15×10^7
django-rest-...	0	3	0.07	4.92×10^4	7.37×10^6
electrum	-	2	-	7.57×10^3	6.49×10^4
Flexget	2	4	0.43	1.70×10^4	4.60×10^6
fonttools	-	-	⊥	1.93×10^4	8.19×10^5
graphene	-	1	-	3.66×10^2	1.19×10^4
hydra	839	952	0.88	3.48×10^4	5.44×10^5
hypothesis	4071	7401	0.55	2.07×10^6	3.41×10^7
ipython	112497	118869	0.95	1.56×10^5	1.78×10^5
kombu	44	63	0.70	8.20×10^3	7.42×10^4
libcloud	984	1686	0.58	1.42×10^5	5.23×10^6
loguru	3	26	0.13	3.77×10^3	1.56×10^5
mitmproxy	90	735	0.12	1.13×10^4	7.68×10^4
Pillow	23	26	0.88	4.57×10^4	4.82×10^5
prefect	103	230	0.45	5.29×10^5	2.19×10^7
PyGithub	2703	2712	1.00	1.45×10^4	7.89×10^4
pyramid	262	383	0.68	4.34×10^3	3.15×10^5
requests	-	-	⊥	6.63×10^3	1.50×10^5
salt	50	65	0.78	2.96×10^4	1.34×10^6
scikit-image	5887	5890	1.00	6.30×10^6	3.19×10^7
seaborn	5	7	0.72	8.31×10^4	1.02×10^6
setuptools	4	4	1.00	1.76×10^4	2.89×10^5
sunpy	-	9	-	1.48×10^5	1.60×10^6
tornado	-	-	⊥	3.05×10^3	9.35×10^4
urllib3	-	-	⊥	7.90×10^3	2.26×10^5
xonsh	9442	9459	1.00	1.15×10^5	1.73×10^6
Overall	182538	194457	0.94	1.30×10^7	1.73×10^8

detection performance that is far greater than the Extra Trees classifier alone.

Figure 5.8b and Table 5.8b show the results for CANNIER+IDFCLASS. As shown by the figure, the general picture is similar to CANNIER+RERUN but somewhat attenuated. The reduction in time cost from original IDFCLASS to balanced CANNIER+IDFCLASS is 84%, slightly less than that for CANNIER+RERUN. In addition, the difference in MCC between the balanced CANNIER+IDFCLASS point (0.97) and the pure machine learning point (0.71) is slightly less significant. The per-project MCC is broadly consistent, as shown by the table. The overall implications of these results are the same as before, namely that applying CANNIER to IDFCLASS sacrifices a minimal degree of detection performance for a considerable reduction in time cost.

Figure 5.8c and Table 5.8c give the results for CANNIER+PAIRWISE. Again, the overall story is similar to the two prior techniques. In this case, the drop in time cost between original PAIRWISE and balanced CANNIER+PAIRWISE is the greatest at 92%. Furthermore, the true-positive rate (TPR) at the point representing balanced CANNIER+PAIRWISE is very high at 0.94. Yet, the table shows that the per-project detection performance varies significantly, far more than the previous two techniques. This could be explained by the relatively high variance in the per-project detection performance of the machine learning pipeline for the Polluter-vs-Rest problem (see Table 5.7b).

Conclusion for RQ4 When applied to RERUN, IDFCLASS, and PAIRWISE, CANNIER is able to reduce time cost by an average of 88% at the expense of only a minor decrease in detection performance.

5.7 Discussion

5.7.1 RQ1. How effective is machine learning-based flaky test detection?

As shown by Table 5.5, there is not much difference in terms of overall MCC between consecutive pipelines in the top-12 for each classification problem. Nonetheless, there are some patterns that have emerged from our choice of pipeline configurations. For NOD-vs-Rest and Victim-vs-Rest, it appears that Extra Trees is the clear winner for the type of classifier, consistently occupying the top positions in both tables. Extra trees is a more randomised variant of Random Forest, an ensemble model based on decision trees [18, 51, 137, 143]. Both fit individual trees on a random subset of the features from a random sample of the data points from the training data. The major difference between the two models is how nodes in the decision tree are split. Random forest uses an optimal split, whereas Extra Trees uses a random split. The additional randomness introduced by Extra Trees trades increased *bias* for reduced *variance*. Increased bias means the classifier may fail to recognise relationships between feature data and labels, known as *underfitting*. Reduced variance means the classifier may be less sensitive to noise and outliers, avoiding *overfitting*. The fact that Extra Trees was more performant with respect to NOD-vs-Rest and Victim-vs-Rest could suggest that this particular trade-off was more beneficial when tackling these two problems, compared to NOD-vs-Victim and Polluter-vs-Rest. The reason for this however would require further investigation.

The pipelines with more trees tended to yield greater detection performance than those of the same classifier type and balancing but with fewer trees. This is expected, since the motivation behind Random Forest and Extra Trees is to fit decision trees with decoupled prediction errors, such that taking an average of their individual predictions leads to some errors cancelling out. Therefore, it stands to reason that more trees would lead to greater performance. Of course, increasing the number of trees can only improve the classifier up to a point — and, moreover, there are some instances in our results where more trees did not lead to better performance.

Plain SMOTE (without additional underbalancing) appeared to yield better pipelines compared to SMOTE+ENN and SMOTE+Tomek. Recall from Section 5.5.2 that SMOTE [23] synthetically increases the number of data points in the minority class via interpolation. However,

the combination of SMOTE with additional underbalancing techniques produces both synthetic members of the minority class but also discards some members of the majority class. It could be that the removal of real data points was detrimental to the performance of the pipelines that used these techniques, though further investigation would be required to be sure.

Table 5.6b shows the per-project and overall results of the best pipeline for the NOD-vs-Victim problem. There is a fairly significant difference between the overall MCC of 0.69 and the per-project mean MCC of 0.55. Recall that `CANNIER-Framework` calculates the overall MCC from the sum of the per-project confusion matrix category frequencies. This disparity is probably caused by the individual results for `IPYTHON` and `AIRFLOW` having a disproportionate impact on the overall result since they have significantly more victim flaky tests than the other subject projects (see Table 5.3). This is also seen in Table 5.7b for `Polluter-vs-Rest`, though in this case the difference between the mean and overall MCC is much larger. Once again, this is likely due to the influence of individual projects with relatively many polluters.

The per-project MCC varies quite considerably, with a standard deviation ranging from 0.22 to 0.34 across the four problems. One would expect that projects with fewer flaky tests would have a poorer MCC than those with more, simply because they have fewer positive examples to train the classifier. However, our results do not appear to show this trend. Therefore, further investigation is required to fully understand why the MCC for some projects is so much greater than that of others.

5.7.2 RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

Our conclusion for RQ2, as illustrated by Figure 5.5, is that increasing the sample size to produce the mean feature vectors increases the overall MCC of the best pipeline for the four flaky test classification problems. This is not surprising, given how the literature has already established a degree of non-determinism in some of the dynamic features in Table 5.1 [70, 139, 160]. What is more interesting is how weak the effect on MCC appears to be, despite being clearly positive, as illustrated by the very small gradient of the line of best fit. Despite this, at the point representing balanced `CANNIER` for all three flaky test detection techniques in RQ4, the number of samples to produce the mean feature vectors (n_F) is fairly high (15, 14, and 9 for `CANNIER+RERUN`, `CANNIER+IDFCLASS`, and `CANNIER+RERUN`, respectively). This suggests that the added time cost of performing the extra feature measurements may be a worthwhile trade-off for the increased detection performance.

5.7.3 RQ3. What contribution do individual features have on the output values of machine learning classifiers for detecting flaky tests?

Figure 5.6 gives the SHAP value beeswarm plots based on the best pipelines for the four flaky test classification problems. These visualise the contribution of the 18 features in Table 5.1 towards the output value of the classifier for a given test case. It is important to remember that Random Forest and Extra Trees are not causal models and therefore it is not appropriate to infer causality by applying SHAP without considering confounding [189]. Furthermore, as demonstrated by our results for RQ1, the detection performance of the classifiers is limited and therefore the SHAP values may not even offer a reliable insight into the correlations between the feature values and the probability of a test case being flaky. Despite this, some of our findings support general intuition and the consensus of the flaky test literature.

For the `NOD-vs-Rest` problem, I found that `Wait Time` appears to contribute positively to the `Extra Trees` classifier output (towards predicting `NOD flaky`) when its value is high and negatively when low. This feature measures the elapsed wall-clock time spent waiting for input/output (I/O) operations to complete. Many empirical studies have pointed to “asynchronous waiting” as a leading cause of `NOD flaky tests` [37, 89, 105, 136], where a test case waits for an insufficient

```
def test_create_pool(self):
    pool = self.client.create_pool(name='foo', slots=1, description='')
    self.assertEqual(pool, ('foo', 1, ''))
    self.assertEqual(self.session.query(models.Pool).count(), 2)
```

(a) This test case from the AIRFLOW project [200] has an AST depth of 1.

```
def test_top_level_return_error(self):
    tl_err_test_cases = self._get_top_level_cases()
    tl_err_test_cases.extend(self._get_ry_syntax_errors())

    vals = ('return', 'yield', 'yield from (_ for _ in range(3))',
            dedent('''
                def f():
                    pass
                return
            '''),
            )

    for test_name, test_case in tl_err_test_cases:
        # This example should work if 'pass' is used as the value
        with self.subTest((test_name, 'pass')):
            iprc(test_case.format(val='pass'))

        # It should fail with all the values
        for val in vals:
            with self.subTest((test_name, val)):
                msg = "Syntax error not raised for %s, %s" % (test_name, val)
                with self.assertRaises(SyntaxError, msg=msg):
                    iprc(test_case.format(val=val))
```

(b) This test case from the IPYTHON project [242] has an AST depth of 5.

Figure 5.9: Two test cases with different values for the AST depth feature. This feature measures the maximum depth of nested program statements.

amount of time for an asynchronous operation, such as I/O, to complete. I also found Context Switches and Max. Children to have a similar contribution pattern. Both of these features are associated with concurrency, another leading cause of flakiness as attested by the same studies. Furthermore, Read Count and Write Count, that measure the number of times the filesystem performed input and output respectively, also appear to contribute positively to the classifier output when high and negatively when low. Previous work has identified I/O itself as a cause of flaky tests [105], but this behaviour could also be related to asynchronous waiting, since Wait Time is time spent waiting for I/O and could correlated with Read Count and Write Count.

For NOD-vs-Rest and NOD-vs-Victim, Run Time has a positive contribution when high and a negative contribution when low and ranks highly in terms of overall contribution (i.e., the mean absolute SHAP value). In their evaluation of FLAKEFLAGGER, Alshammari et al. [8] also found the execution time of test cases to be correlated with the probability of being NOD flaky. However, they were unable to establish any casual link. For the Victim-vs-Rest problem, Write Count, Read Count, and Wait Time seem to contribute have a similar contribution pattern, but to varying degrees of consistency. Since these features are associated with I/O, this correlation could be explained by the relationship between filesystem activity and victim flaky tests established in previous studies (e.g., [15, 17, 47, 105, 175]).

Seven of the 18 features are static, meaning they are based on the test case code and do not require a test case execution to measure. One of these is AST Depth that measures the maximum depth of nested program statements. Figure 5.9 compares two test cases with different values for the AST depth feature. In terms of mean absolute SHAP value, AST Depth was the most impactful for the NOD-vs-Rest problem. While no previous study has examined the relationship between AST Depth and flakiness, intuitively one might expect a high AST Depth to be associated

with a higher chance of flakiness. This is simply because a test case with a higher AST Depth is likely to be more complex and therefore offer more opportunities for flakiness to arise. The beeswarm plot for NOD-vs-Rest appear to broadly support this notion yet the plots for the other problems do not indicate a clear relationship. This suggests that AST Depth may be correlated with the probability of a test case being NOD flaky.

There appear to be some tentative relationships between the contribution patterns of features for the four problems. For NOD-vs-Rest and NOD-vs-Victim, the contribution of Run Time, Read Count, Context Switches, Write Count, and Max. Children are broadly positive when high and negative when low. This could be due to the positive class being the same for both problems and the negative class of NOD-vs-Victim being a subset of the negative class of NOD-vs-Rest. Moreover, the contribution pattern of the features for the NOD-vs-Victim differs significantly that of the Victim-vs-Rest problem. As I reported in Section 5.6.3, the Maintainability and Cyclomatic Complexity features appear to have nearly opposite contribution patterns between the two problems. This is expected, because the positive class of Victim-vs-Rest is the negative of NOD-vs-Victim, and the negative class of Victim-vs-Rest is a superset of the positive of NOD-vs-Victim.

It is clear from Figure 5.7 that dropping the less impactful features (in terms of mean absolute SHAP value) has little impact on the detection performance of the best pipeline for each problem. Since the time to fit a Random Forest/Extra Trees classifier grows linearly with the number of features, this is a useful result for expediting the training stage. This is not directly relevant to the conclusions of this study however, as I am not concerned with the time cost of classifier training since that is performed off-line from the perspective of a developer using the CANNIER approach.

5.7.4 RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

I presented CANNIER+iDFCLASS as a drop-in replacement for the Classification stage of iD-FLAKIES. In theory, the combination of the NOD-vs-Rest and Victim-vs-Rest classifiers could be a substitute for the entire iD-FLAKIES pipeline. This could be realised as CANNIER+iD-FLAKIES, a multi-classifier approach with a multi-label output: NOD, Victim, or Rest (non-flaky). In practice, the difficulty arises when either of the classifiers are ambiguous for a given test case. To delegate the prediction for such a test case to iD-FLAKIES in this hypothetical scenario, CANNIER+iD-FLAKIES would need to rerun the entire test suite in different orders until the test case fails or the upper-limit is reached. This corresponds to the Running stage of iD-FLAKIES. As with the single-classifier CANNIER+iDFCLASS given in the study, it would then execute the prefix of the failing test order, representing the Classification stage of iD-FLAKIES. Naturally, with even a handful of ambiguous cases, the hypothetical multi-classifier CANNIER+iD-FLAKIES would be unlikely to noticeably reduce the time cost of the Running stage, but would reduce the time cost of the Classification stage in the same way as the existing single-classifier CANNIER+iDFCLASS. Therefore, the benefit of CANNIER+iD-FLAKIES is effectively the same as CANNIER+iDFCLASS, since the latter makes no attempt to expedite the Running stage. For these reasons, I opted to focus on CANNIER+iDFCLASS due to its simplicity and the fact that it would require fewer modifications to iD-FLAKIES to implement.

As shown in Figure 5.8a and Table 5.8, for the point representing balanced CANNIER+RERUN, the lower-threshold (ω_l) is very low at 0.07 and the upper-threshold (ω_u) is at its maximum value of 1.01. The latter means that there effectively is no upper-threshold on the predicted probability (see the second clause of Equation 5.4). Figure 5.10 illustrates the distribution of predicted probabilities for test cases in, and gives the frequencies of, each confusion matrix category, for each of the four flaky test classification problems. I produced this figure from the results of RQ1, such that the figure for each classification problem corresponds to its respective table in Tables 5.6 and 5.7. Figure 5.10a focuses on the NOD-vs-Rest problem. The distribution for true-negatives (TN) is focused largely around 0 and represents the vast majority of test cases. Furthermore, the distribution for false-negatives (FN) appears highly separable

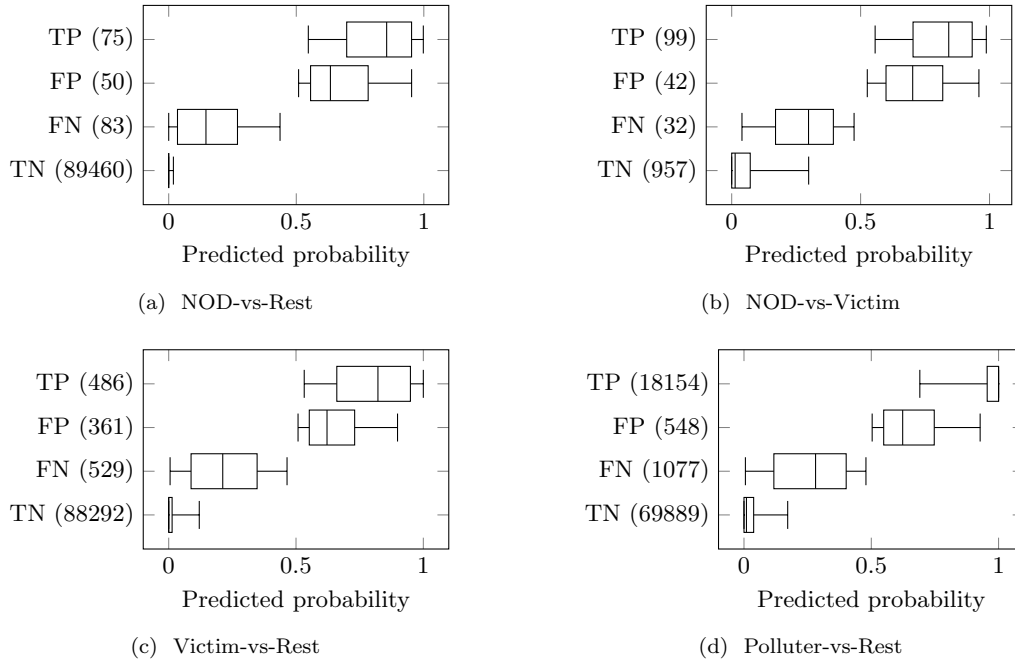


Figure 5.10: The distribution of predicted probabilities for test cases in, and the frequencies of, each confusion matrix category, for each of the four flaky test classification problems. The data is based on the best pipelines from RQ1. Whiskers represent the range from the 5th to the 95th percentile and boxes represent the 25th to the 75th. Middle lines represent the median (50th).

from true-negatives. This might explain why ω_l is so low, because it means CANNIER+RERUN labels most true-negative test cases as negative and prevents them from being delegated to RERUN, significantly reducing time cost. It also means CANNIER+RERUN labels only a handful of false-negatives as negative, limiting the reduction in detection performance. The distribution for true-positives (TP) is clearly different from false-positives (FP) but not as easily separable. However, there are few test cases in both categories relative to true-negatives. Therefore, by setting ω_u to its maximum value, CANNIER+RERUN makes no false-positive predictions, ensuring no decrease in detection performance at the expense of a minor increase in time cost. This could explain why there are no false-positive predictions in Table 5.8.

Figure 5.10b illustrates the distribution of predicted probabilities for NOD-vs-Victim. The situation for this problem and the thresholds for the point representing balanced CANNIER+iDFCLASS is very similar to NOD-vs-Rest and CANNIER+RERUN. The biggest difference is that the frequency of the true-negative category for NOD-vs-Victim is two orders of magnitude smaller than that for NOD-vs-Rest. The distribution for true-negatives also spreads much further into the distribution for false-negatives. This may explain why the lower-threshold for CANNIER+iDFCLASS is greater at 0.18 and why the reduction in time cost from iDFCLASS to CANNIER+iDFCLASS is smaller.

Figure 5.10c and 5.10d are for Victim-vs-Rest and Polluter-vs-Rest respectively. Once again, the overall picture is similar for both problems. That is, the true-negative category contains the vast majority of test cases and its distribution is broadly separable from the false-negative category. This explains why the victim-threshold (ω_V) and polluter-threshold (ω_P) for the balanced CANNIER+PAIRWISE point are low at 0.06 and 0.08 respectively. Uniquely for Polluter-vs-Rest, the true-positive distribution appears very distinct from the false-positive distribution. Perhaps because this problem has significantly more positive examples in the dataset compared to the other problems, the machine learning classifier can discern unseen positive cases with greater confidence.

5.7.5 Implications

Researchers

Our findings for RQ1 extend the existing body of work in machine learning-based flaky test detection into the detection of polluter test cases. Identifying polluters is vital for mitigating test-order dependencies [91, 119, 141] and so our results demonstrate the wider applicability of machine learning classifiers for tackling flaky tests. Our results for RQ2 (and supported by RQ4) demonstrate that using mean feature vectors can improve the detection performance of machine learning classifiers. I therefore suggest that researchers consider the implications of this when evaluating machine-learning based techniques that use dynamic features. Our results for RQ3 tentatively identify correlations between test case metrics and the probability of a test case being flaky. This is an important foundation for future work in elevating flaky test detection techniques to comprehensive flaky test root causing techniques, a vital intermediate step towards automated flaky test repair. While such root causing and repair techniques exist [88, 151, 162], they are expensive and limited in scope.

Developers

Our findings for RQ4 demonstrate that CANNIER is a “best of both worlds” approach between rerunning-based and machine learning-based flaky test detection. As shown by Figure 5.8, CANNIER reduces time cost by an average of 88% across the three rerunning-based techniques while maintaining good detection performance. For developers, this means not having to trade high time cost for limited detection performance. Furthermore, while I used the knee-point of the Pareto front to represent CANNIER in our evaluation, developers could customise the approach towards lower time cost or greater detection performance by selecting a different point.

5.8 Related Work

Luo et al. [105] performed one of the earliest empirical studies of test flakiness. Using 51 projects of the Apache Software Foundation as subjects, they classified 201 commits that repaired flaky tests into 10 categories based on the cause of the flakiness. The most common cause they identified was related to waiting for asynchronous operations. For example, a test case that launches a thread to perform input/output (I/O) and waits a fixed amount of time for it to finish may fail when it takes longer than expected. One of the findings for RQ3 was that the amount of time spent waiting for I/O operations to complete was positively correlated with the probability of a test case being NOD flaky.

Gruber et al. [59] repeatedly executed the test suites of 22,352 open-source projects and automatically identified 7,571 flaky tests. Like this study, these projects were primarily written in the Python programming language. They randomly sampled 100 NOD flaky tests in their dataset to classify their causes using the categories introduced by Luo et al. [105]. Unlike Luo et al., they found causes related to networking and randomness to be the most prevalent.

Bell et al. [16] presented an automated technique, called DEFLAKER, for detecting NOD flaky tests. The key advantage of DEFLAKER over RERUN is that it does not require repeated test case executions. Instead, the technique takes advantage of a project’s history in a version control system. When a test case that passed on a previous version of the software now fails, and does not cover modified code, DEFLAKER labels it as flaky. Naturally, DEFLAKER requires a test suite run with code instrumentation to measure coverage. Detecting flaky tests using Extra Trees models with CANNIER-Framework also requires an instrumented run to measure coverage and the other metrics in Table 5.1. In both cases, this test suite run introduces time overhead. However, DEFLAKER requires a run every time a change is made, whereas CANNIER-Framework requires at least one to produce encodings for each test case that would likely remain relevant over a series of changes. Furthermore, DEFLAKER can only detect flaky tests after they fail. In contrast, the models trained by the CANNIER-Framework can detect flaky tests preemptively.

Pinto et al. [128] and Bertolino et al. [159] both presented machine learning-based flaky test detection techniques based purely on static features of the test case code. Both techniques encoded test cases using a bag-of-words approach. This represents test cases as sparse vectors where each element corresponds to the frequency of a particular identifier or keyword in its source code. Pinto et al. used additional static features such as the number of lines of code. Bertolino et al. used a *k-nearest neighbour* classifier [80] for the machine learning classifier and Pinto et al. evaluated a range of different models, including Random Forest. They found Random Forest to yield the best detection performance, of which I use the Extra Trees variant in this study, having found it to be the most effective in Chapter 4. Alshammari et al. [8] presented FLAKEFLAGGER, a detection technique using a Random Forest classifier and encoding test cases with a feature set containing a mixture of static and dynamic test case metrics. Their evaluation showed that their feature set offered a 347% improvement in overall F1 score compared to Pinto et al.’s purely static feature set at the cost of a single instrumented test suite run to measure the dynamic features. For this reason, I included both static and dynamic test metrics in the feature set instead of relying on purely static features.

Shi et al. [141] presented IFIXFLAKIES, a technique for automatically generating patches for victim flaky tests. Their approach uses delta-debugging [172] to identify a victim’s polluters and other test cases that may contain the statements needed to repair the victim, known as *cleaners*. CANNIER+PAIRWISE could provide a drop-in replacement for this aspect of IFIXFLAKIES. However, I cannot say for certain if it would be faster than using delta-debugging because I have not yet evaluated it in this context.

Lam et al. [90] presented IDFLAKIES, a technique for detecting flaky tests and classifying them as either NOD or Victim. The overall process involves repeatedly executing a test suite in a modified order (e.g., shuffled) to identify flaky test cases. Following this, the tool enters a Classification stage where it attempts to determine the category of each flaky test. In this study, I evaluated the application of CANNIER to the Classification stage of this tool (CANNIER+IDFCCLASS). The empirical results demonstrated that CANNIER was able to significantly reduce the execution time overhead of the Classification stage at minimal detriment to its detection performance.

5.9 Conclusions

This study expanded the existing work on machine learning-based flaky test detection and introduced CANNIER, an approach for significantly reducing the time cost of rerunning-based detection techniques by combining them with machine learning classifiers. Initially, using a variety of machine learning pipelines and a feature set of 18 static and dynamic test case metrics, I performed a baseline evaluation of machine learning-based detection on a dataset of 89,668 test cases from 30 Python projects. I evaluated their performance with respect to detecting NOD flaky tests, victim flaky tests, and polluter test cases. The results suggested that the performance of the machine learning classifiers was lacklustre and variable between projects. I then went on to investigate the impact of mean feature vectors on machine learning-based flaky test detection. I identified a positive relationship between the sample size to produce the mean feature vectors and the detection performance of the machine learning classifier. In the interest of model explainability, I applied the SHAP technique [103] to quantify the contribution of each individual feature to the output value of the classifier. While this technique can only reveal correlations and is not appropriate for inferring causality, I made several findings that support both the general intuition of developers and results from the flaky test literature. Finally, I evaluated CANNIER’s impact on three rerunning-based methods for flaky test detection: RERUN, the Classification stage of IDFLAKIES, and PAIRWISE. I found that CANNIER was able to significantly reduce time cost at the expense of only a minor decrease in detection performance.

While the evaluation in this chapter does indeed demonstrate that CANNIER is the “best of both worlds” between using classifiers and rerunning test cases for detecting flaky tests, it could still impose a significant cumulative time cost over repeated use as developers introduce new test cases. Hypothetically, a classifier trained only on the test cases of a specific project could provide

better predictions for the future test cases of that project than a classifier trained on the test cases of many projects. This is because such a classifier would not have to generalise to other projects. While a project-specific classifier may accurately detect future flaky tests very quickly, providing the training data would be a problem. This is because one would need an efficient approach to detect the existing flaky tests in the project to provide the training labels for the classifier. Because the classifier is effectively using this approach as a ground truth it needs to be very accurate, but the time cost cannot be too high otherwise the long-term benefits of using the project-specific classifier will be significantly eroded.

Chapter 6

Automatically Training Project-Specific Machine Learning Models to Detect and Classify Flaky Tests

The contents of this chapter is based on research currently under review at the International Conference on Automated Software Engineering 2023.

6.1 Introduction

Researchers have introduced automated techniques for the detection of flaky tests based on exhaustively rerunning test cases, reasoning that flaky tests will exhibit inconsistent outcomes after enough executions [16, 59, 90, 92]. The downside is the time cost of executing test cases hundreds or even thousands of times. For developers working with large test suites, this cost can be prohibitive. The intractability of rerunning-based techniques has pushed research towards machine-learning based detection. A typical methodology is to rerun the test suites of a corpus of projects many times to produce a set of ground-truth labels, with which to train a machine learning classifier in combination with features regarding every test case [8, 128, 133, 159]. While evaluations based on cross-validation show promising results, researchers have established that classifiers perform poorly when evaluated on test cases from projects that were not part of their training data [20, 41]. Since the idea is that a developer takes the pre-trained classifier and applies it to their own projects, this poor inter-project generalisability seriously threatens their usefulness.

In this study, I present the FLAKEFRIEND technique. It is capable of not only detecting and classifying existing flaky tests in a project, but also of producing reusable project-specific machine learning classifiers able to provide fast and accurate predictions for future test cases in that project. These classifiers can be viewed as a “friend” of the test suite, warning developers of flaky tests early on to avoid them accumulating [180]. I developed a concrete implementation to enable an extensive empirical evaluation involving 63,090 test cases from 10 open-source Python projects. The implementation and evaluation draws strongly on the findings of Chapters 3 and 5 to ensure relevance to developers and practicality respectively. I found that FLAKEFRIEND is able to detect existing flaky tests in a project in just 20% of the time taken by exhaustive rerunning on average. I found that the machine learning classifiers produced by FLAKEFRIEND for a project can detect future flaky tests in that project with a mean Matthews Correlation Coefficient (MCC), a reliable metric for evaluating a classifier [24], of 0.73. This is significantly better than classifiers pre-trained on the test cases of other projects, that I found to be consistently poor with a mean MCC of just 0.18. This supports previous findings that classifiers for detecting flaky tests do not generalise

well between projects [20, 41]. I also found that FLAKEFRIEND offers considerable savings in cumulative time cost over repeated use when detecting flaky tests in evolving test suites compared to a previous state-of-the-art approach.

In summary, the main contributions of this study are:

1. **Technique (Section 6.2):** A novel technique capable of detecting and classifying existing flaky tests in a project, and of producing machine learning classifiers to provide fast and accurate predictions for future test cases.
2. **Dataset (Section 6.3):** Over several months of compute time, I collected a large dataset of test run information regarding 10,000 executions of 63,090 test cases. This is freely available in the replication package [205] to enable future testing research.
3. **Evaluation (Section 6.4):** An extensive empirical evaluation involving 10 open-source projects shows how FLAKEFRIEND addresses the drawbacks of both rerunning-based and machine learning-based flaky test detection.

6.2 FLAKEFRIEND

Machine learning classifiers for detecting flaky tests perform poorly when evaluated on projects that were not part of their training data [20, 41]. Their performance could be improved by some oracle on their predicted probabilities, providing binary “flaky or not flaky” labels for every test case. It would be expected to impose a small time cost, relative to exhaustive rerunning, for significantly greater detection performance. While this may provide a “best of both worlds” solution between applying classifiers alone and exhaustive rerunning, it may still impose a significant cumulative time cost over repeated use as developers introduce new test cases. The innovation behind FLAKEFRIEND is to use an existing *project-agnostic model*, a classifier pre-trained on many projects, in combination with an oracle to provide *project-agnostic labels*, binary labels indicating flakiness, for the existing test cases of an unseen project, with which to train a *project-specific model* for that project. Because the project-specific model does not need to generalise to other test suites, it should produce high-quality *project-specific labels* for the future test cases of that project without the support and added time cost of the oracle. Therefore, the time cost of applying the project-specific model is minimal and the majority of the cumulative time cost of FLAKEFRIEND is paid when the test suite is smaller, as opposed to over time as the test suite grows.

FLAKEFRIEND requires a test suite as input, a set of n test cases t_1, t_2, \dots, t_n . At the point in time when FLAKEFRIEND is first applied, the test suite is split between existing and future test cases. The index of the first future test case is k , such that t_1, t_2, \dots, t_{k-1} are existing and t_k, t_{k+1}, \dots, t_n are future. Figure 6.1 provides an overview of the technique and an example with three existing test cases and three future tests cases. Where α_i is the time cost of extracting a feature vector for a test case t_i to feed into the various models, and β_i is the time cost imposed by the oracle of acquiring project-agnostic labels for it, the cumulative time cost of applying FLAKEFRIEND in this example is $\sum_{i=1}^6 \alpha_i + \sum_{i=1}^3 \beta_i$. This results in every test case having predicted labels, the project-agnostic labels for the three existing test cases and the project-specific labels for the three future test cases. Generalizing, the initial cost paid by a user of FLAKEFRIEND is $\sum_{i=1}^{k-1} (\alpha_i + \beta_i)$. For every future test case t_i , the user pays α_i , the cost of extracting a feature vector to feed into the project-specific models to get project-specific labels. If the user were to obtain project-agnostic labels instead, they would pay the additional cost imposed by the oracle, thus paying $\alpha_i + \beta_i$ for each future test case. The recovery of β_i can result in considerable savings as the test suite evolves. As part of the empirical evaluation, I found that FLAKEFRIEND saved an average of 118 days of single-core cumulative time cost after doubling the size of the test suite (see Section 6.5.3). Therefore, the more complex process involved in FLAKEFRIEND is clearly justified.

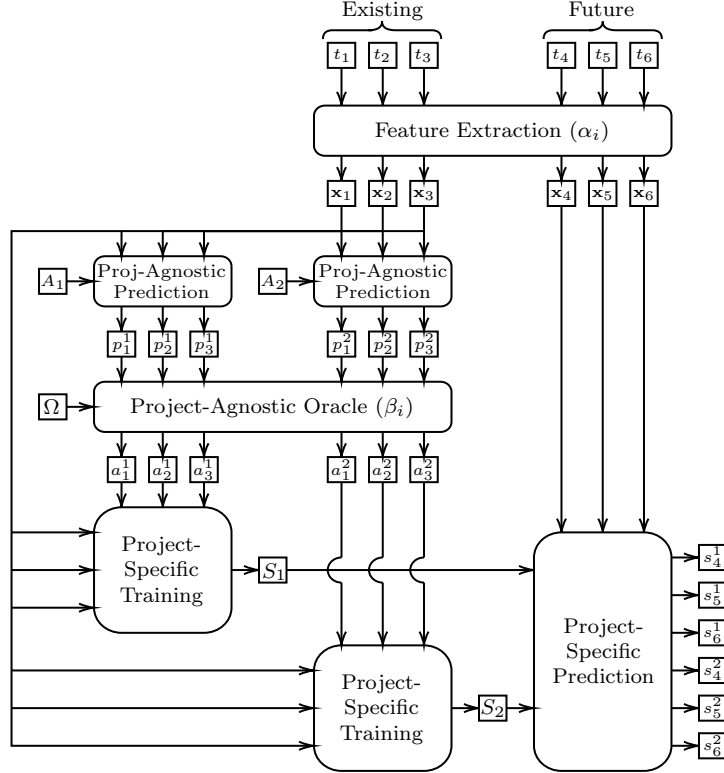


Figure 6.1: An overview of FLAKEFRIEND. The inputs are a set of n test cases (t_1, t_2, \dots, t_n) , a set of m project-agnostic models (A_1, A_2, \dots, A_m) , and parameters for the Project-Agnostic Oracle stage (Ω) . Where k is the index of the first future test case, t_1, t_2, \dots, t_{k-1} are existing test cases and t_k, t_{k+1}, \dots, t_n are future test cases. The primary outputs are a set of $(k-1) \times m$ project-agnostic labels $(a_1^1, a_2^1, \dots, a_{k-1}^m)$, a set of m project-specific models (S_1, S_2, \dots, S_m) , and a set of $(n-k+1) \times m$ project-specific labels $(s_k^1, s_{k+1}^1, \dots, s_n^m)$. The intermediate outputs are a set of n feature vectors $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$ and a set of $(k-1) \times m$ predicted probabilities $(p_1^1, p_2^1, \dots, p_{k-1}^m)$. For processing an individual test case t_i , the Feature Extraction and Project-Agnostic Oracle stages contribute a time cost of α_i and β_i respectively to the cumulative time cost of applying FLAKEFRIEND. In this example, $n = 6$, $m = 2$, $k = 4$, and the cumulative time cost is $\sum_{i=1}^6 \alpha_i + \sum_{i=1}^3 \beta_i$. See Sections 6.2.1 to 6.2.5 for further explanation.

6.2.1 Feature Extraction

The first stage of the technique is to extract a feature vector \mathbf{x}_i to represent every test case t_i . The technique places no explicit requirements on this stage since the nature of the features are dependent on the type of machine learning classifiers used in the later stages. However, it may involve executing the test cases, potentially with run-time instrumentation. Therefore, FLAKEFRIEND expects a *feature cost* α_i to be associated with producing the feature vector for a test case t_i . In Figure 6.1, the test suite consists of six test cases in total, and the output of the Feature Extraction stage is six feature vectors.

6.2.2 Project-Agnostic Prediction

FLAKEFRIEND requires m pre-trained project-agnostic models as input A_1, A_2, \dots, A_m . The first, A_1 , should predict the probability $p_i^1 \in [0, 1]$ that a test case t_i is flaky. The subsequent models should predict the probability that a test case is a specific category of flaky. In other words, where $y_i^1 \in \{0, 1\}$ is a *ground-truth label* indicating if t_i is flaky (unknown to the user of the technique), A_1 estimates $P(y_i^1 = 1)$. For $j > 1$, where y_i^j is a ground-truth label indicating if t_i is a specific category of flaky, A_j estimates $P(y_i^j = 1)$.

Much research in the field of flaky tests has enumerated many different categories of flakiness (see Table 2.4). These are useful for developers to know when attempting to repair them [56]. FLAKEFRIEND expects the project-agnostic models to have been pre-trained on a set of projects that does not include the project that the input test suite is from. For each project-agnostic model, FLAKEFRIEND applies the Project-Agnostic Prediction stage to the existing test cases only. In Figure 6.1, the technique applies two project-agnostic models to the three existing test cases, resulting in six predicted probabilities.

6.2.3 Project-Agnostic Oracle

The purpose of the Project-Agnostic Oracle stage is to take each predicted probability p_i^j from the previous stage and convert it into a project-agnostic label $a_i^j \in \{0, 1\}$ with equivalent semantics to its corresponding ground-truth label y_i^j . To ensure consistent labelling, the only restriction FLAKEFRIEND places on this stage is that $a_i^j = 1 \implies a_i^1 = 1$ for $j > 1$ must always hold, since this also naturally applies to the ground-truth labels. In other words, the oracle should never label a test case t_i as a specific category of flaky ($a_i^j = 1$) if it does not also label the test case as flaky ($a_i^1 = 1$). The technique expects an *oracle cost* β_i to be associated with processing t_i , that is, converting every p_i^j to a_i^j . Furthermore, FLAKEFRIEND can take an additional input Ω representing the parameters of the oracle, though the technique does not specify what they may be. Figure 6.1 shows the six predicted probabilities for the three existing test cases becoming six project-agnostic labels.

6.2.4 Project-Specific Training

Given $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{k-1}$ as feature vectors from the Feature Extraction stage and $a_1^j, a_2^j, \dots, a_{k-1}^j$ as training labels from the previous stage, the Project-Specific Training stage produces a project-specific model S_j to correspond with a project-agnostic model A_j . In other words, FLAKEFRIEND uses A_j via the oracle to train S_j to detect the same type of flakiness as A_j exclusively within the context of the input project. The details of the training procedure depends on the type of machine learning classifier, and so FLAKEFRIEND places no restrictions on how this is performed. Figure 6.1 shows FLAKEFRIEND using the three project-agnostic labels from A_1 to train S_1 and the three project-agnostic labels from A_2 to train S_2 .

6.2.5 Project-Specific Prediction

This stage takes the project-specific models from the Project-Specific Training stage and applies them to the future test cases. For a future test case t_i , a project-specific model S_j predicts a project-specific label $s_i^j \in \{0, 1\}$. These labels are semantically equivalent to their project-agnostic counterparts. At this stage, FLAKEFRIEND enforces the same labelling restriction as it did in the Project-Agnostic Oracle stage, namely that $s_i^j = 1 \implies s_i^1 = 1$ for $j > 1$ must always hold. In Figure 6.1, the technique applies the two project-specific models to the three future test cases that were not involved in training, producing six project-specific labels as output.

6.3 Implementation

I implemented FLAKEFRIEND with a plugin to the Python testing framework `pytest` [266] and a collection Python scripts, named `FlakeFriendEvaluator`. These are freely available within the replication package [205]. While FLAKEFRIEND is formulated to operate upon a single test suite and to execute each stage once, the implementation supports multiple subject projects as input and repeats stages over a range of classifier types, feature sets, and parameters to facilitate the evaluation and to improve its generalisability.

Table 6.1: The features collected by the modes of `pytest-FlakeFriend`. (¹Scalar metric feature. ²Sparse boolean feature vector. ³Feature excluded from training project-agnostic models.)

Mode	Features
Static	AST Depth ¹ , External Modules ¹ , Assertions ¹ , Halstead Volume ¹ , Cyclomatic Complexity ¹ , Test Lines of Code ¹ , Maintainability ¹ , Test Function Tokens ²
Dynamic	Static + Elapsed Time ¹ , User Time ¹ , System Time ¹ , Wait Time ¹ , Read Count ¹ , Write Count ¹ , Context Switches ¹ , Max. Threads ¹ , Max. Children ¹ , Max. Memory ¹
DFunc	Dynamic + Function Coverage ^{2,3} , Function Transition Coverage ^{2,3}
DLine	Dynamic + Covered Lines ¹ , Covered Line Tokens ² , Line Coverage ^{2,3} , Line Arc Coverage ^{2,3}
DFL	DFunc + DLine

6.3.1 Feature Extraction

To implement this stage, I created a `pytest` plugin named `pytest-FlakeFriend`. It offers six modes for executing a test suite where five collect test case features (see Table 6.1). Most are metrics but some are sparse boolean vectors, where each element represents an occurrence. In each of these five modes, the plugin records the feature cost α_i of every test case t_i as a time cost in seconds. For each subject project and for each of the five modes, `FlakeFriendEvaluator` launches the plugin inside a Docker container on a high-performance cluster with a 72 hour timeout.

The first feature-collecting mode is named *Static*, in which `pytest-FlakeFriend` statically analyses the test function associated with every test case without executing them. Since the `pytest` framework allows test cases to be parameterised, multiple test cases can share a single test function and would therefore share the same features. Included in the features collected in this mode are the seven static features from `FLAKE16` (see Table 4.1), capturing the complexity of the test function code. `pytest-FlakeFriend` also produces a sparse boolean feature vector for every test case representing the occurrences of tokens in the source code of its test function. This follows previous evaluations of machine learning classifiers for flakiness detection [128, 159].

The second mode is named *Dynamic*. It includes all the analyses of the Static mode but also executes every test case to collect dynamic metrics. Many of the metrics are taken from the evaluation of the `CANNIER` approach for detecting flaky tests (see Table 5.1), with some minor additions. The User Time feature measures the amount of time spent in user-mode code within the process during test case execution. The System Time feature measures the time spent in the kernel, which is typically required for privileged system services.

The third mode, named *DFunc* (Dynamic + Function), subsumes the Dynamic mode. It also produces two sparse boolean feature vectors: Function Coverage and Function Call Coverage. Each element in Function Coverage represents a function in the code under test that could be called, and an element is set if the function is called at least once during test case execution. Each element in Function Transition Coverage represents a possible transition between a caller function and a callee function and is set if the caller calls the callee.

The fourth mode is *DLine* (Dynamic + Line). Like the third mode, it subsumes Dynamic but concerns line coverage rather than function coverage. It introduces a metric, Covered Lines, that is the number of lines in the code under test covered during test case execution. It also introduces three sparse boolean vectors. The Covered Line Tokens feature is similar to Test Function Tokens, but records the tokens that occur in the lines covered by a test case rather than in its test function. The Line Coverage feature is similar to Function Coverage, but each element represents a line that could be covered. The Line Arc Coverage feature, analogous to Function Transition Coverage, records the transitions between lines.

The fifth and final mode, *DFL* (Dynamic + Function + Line), is the combination of the two previous modes and therefore includes the analyses of all modes. This collects the most information about each test case at the cost of the most extensive run-time instrumentation.

6.3.2 Project-Agnostic Prediction

I used two project-agnostic models focusing on non-order-dependent (NOD) flaky tests. The first (A_1) estimates the probability that a test case is NOD flaky. The second (A_2) estimates the probability that a test case is NOD flaky due to the specification of the machine (NODSpec). Developers highlighted this category during the survey in Chapter 3 (see Section 3.3).

Training them requires a ground-truth label y_i^j for every pair of test case t_i and project-agnostic classifier A_j . To collect these, `FlakeFriendEvaluator` executes each subject project’s test suite 10,000 times on a cluster using the sixth mode of `pytest-FlakeFriend`, `Rerun`. Unlike the others, `Rerun` does not collect features but records the pass/fail outcome of every test case t_i as well as its time cost in seconds γ_i^l on each run l . `FlakeFriendEvaluator` launches each run inside a Docker container with the `--cpu`, `--disk-read-bps`, and `--disk-write-bps` options (that limit the CPU and disk speed) set to random values to simulate machines of varying specification [214]. They record the ground-truth label y_i^1 for a test case t_i and project-agnostic model A_1 as positive ($y_i^1 = 1$) if the test case gave an inconsistent outcome. For A_2 , they record the ground truth label y_i^2 as positive if the test case was NOD flaky ($y_i^1 = 1$) and there is a significant difference in the values for any of the Docker options between the passing and failing runs of t_i . For each flaky test, `FlakeFriendEvaluator` performs a Mann-Whitney U test with a 95% interval for each option.

For each subject, `FlakeFriendEvaluator` produces 15 pairs of concrete classifiers for A_1 and A_2 , considering all combinations of the five feature-collecting modes of `pytest-FlakeFriend` and three types of classifier. For their implementation, I used `scikit-learn` [271]. The first type is Extra Trees (ET) [51], a variant of Random Forest [18] with SMOTE preprocessing to balance the classes in the training data [23]. I selected this pipeline due to its performance when detecting flaky tests in recent studies [8] and in Chapters 4 and 5. Since this is designed for scalar features, it cannot use any of the sparse binary feature vectors collected by `pytest-FlakeFriend`. For this setup, `FlakeFriendEvaluator` uses the meta-parameters that led to the best detection performance in the evaluation of CANNIER (see Table 5.5a). The second is K-Nearest Neighbors (KNN) with Random Projection preprocessing to reduce dimensionality, as used by Verdecchia et al. in their study on detecting flaky tests [159]. In contrast with ET, `FlakeFriendEvaluator` only uses the sparse features. In this case, it uses the same meta-parameters as used by Verdecchia et al.. The third type is the combination of the previous two (ET+KNN). `FlakeFriendEvaluator` uses the probability predicted by the KNN classifier as a feature for the ET classifier, allowing it to use all the features in training. In addition to the restrictions imposed by the classifier type, it excludes the Function Coverage, Function Transition Coverage, Line Coverage, and Line Arc Coverage features from training. Because they track coverage within the code under test and are specific to the project the test cases are from, they are inappropriate for a project-agnostic model.

`FlakeFriendEvaluator` trains the classifiers using the test cases from every subject excluding one. Then, it gets probabilities from the trained classifiers for the test cases of the excluded subject. `FlakeFriendEvaluator` repeats this procedure for every subject to collect probabilities for every test case as if its test suite was the input test suite to `FLAKEFRIEND` and was not part of the training data for the project-agnostic models. This methodology has been termed *cross-project validation* in previous work [20].

6.3.3 Project-Agnostic Oracle

To implement this stage, I applied the CANNIER approach (see Section 5.3). That is, to label test cases with a predicted probability of flakiness above some upper-threshold ω_u as flaky, those below some lower-threshold ω_l as non-flaky, and to delegate the rest to some rerunning-based technique. These thresholds are the parameters Ω in this implementation. This leads to the definitions of the project-agnostic labels a_i^1 and a_i^2 for test case t_i , corresponding to A_1 and A_2 :

$$a_i^1 = \begin{cases} y_i^1 & \text{if } \omega_l \leq p_i^1 < \omega_u \\ 1 & \text{if } p_i^1 \geq \omega_u \\ 0 & \text{otherwise} \end{cases} \quad a_i^2 = \begin{cases} y_i^2 & \text{if } \omega_l \leq p_i^1 < \omega_u \\ 1 & \text{if } a_i^1 = 1 \wedge p_i^2 \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \quad (6.1)$$

Table 6.2: The repository names, commit SHAs, and topic classifiers of the subject projects. (`xonsh/xonsh` had no topic classifiers.)

Project	SHA	Topic Classifiers
apache/airflow	0bcdba0	System :: Monitoring
home-assistant/core	b4f1683	Home Automation
dask/dask	a77ce95	Scientific/Engineering, System :: Distributed Computing
fonttools/fonttools	8bc00a6	Multimedia :: Graphics, Multimedia :: Graphics :: Graphics Conversion, Text Processing :: Fonts
HypothesisWorks/hypothesis	173de52	Education :: Testing, Software Development :: Testing
ipython/ipython	15ea1ed	System :: Shells
PrefectHQ/prefect	f6648e4	Software Development :: Libraries
saltstack/salt	4bbdd65	System :: Clustering, System :: Distributed Computing
urllib3/urllib3	64b7f79	Internet :: WWW/HTTP, Software Development :: Libraries
xonsh/xonsh	ebadfac	-

For NOD flakiness, `FlakeFriendEvaluator` uses the value of the ground-truth label, representing the outcome of exhaustive rerunning, for the project-agnostic label if the probability from A_1 is between the thresholds. If the probability is greater than or equal to the upper-threshold, it labels the test case as flaky. For NODSpec flakiness, `FlakeFriendEvaluator` again uses the ground-truth label if the probability from A_1 is between the thresholds. This is because, due to the nature of the rerunning-based technique described in the previous section, the ground-truth labels for both NOD and NODSpec are acquired together for the same time cost. Otherwise, if `FlakeFriendEvaluator` previously labelled the test case as NOD flaky, and the probability from A_2 indicates that the test case is more likely than not NODSpec flaky, then it labels the test case as such. The time cost of acquiring the ground-truth labels for a test case between the thresholds is the total time cost of executing it 10,000 times. Therefore, the oracle cost for test case t_i is:

$$\beta_i = \begin{cases} \sum_{l=1}^{10,000} \gamma_i^l & \text{if } \omega_l \leq p_i^1 < \omega_u \\ 0 & \text{otherwise} \end{cases} \quad (6.2)$$

Since this stage requires values for the thresholds (ω_l, ω_u) , `FlakeFriendEvaluator` repeats it over $\{(0.04a, 0.04b) | a \in \{0, \dots, 27\}, b \in \{a, \dots, 27\}\}$. This set represents a uniform sample of the space of valid threshold values such that $\omega_l \leq \omega_u$. It contains 378 pairs of threshold values, and therefore `FlakeFriendEvaluator` produces 378 sets of project-agnostic labels for each of the 15 sets of predicted probabilities from the Project-Agnostic Prediction stage for each subject project, making 5,670 per project.

6.3.4 Project-Specific Training

Since there are two project-agnostic models in this implementation, there must also be two project-specific models S_1 and S_2 per subject project. For each set of project-agnostic labels from the Project-Agnostic Oracle stage and for each classifier type, `FlakeFriendEvaluator` uses the feature vectors associated with the set of project-agnostic labels, and the project-agnostic labels themselves, to train ten pairs of concrete classifiers of the classifier type for the two project-specific models. This is the result of ten-fold cross-validation over the test cases of the project associated with the set of project-agnostic labels. The test cases in the training portion of each fold represent the existing test cases at a given point in time and those in the testing portion represent future test cases. Unlike when training the project-agnostic models, `FlakeFriendEvaluator` does not exclude the Function Coverage, Function Transition Coverage, Line Coverage, and Line Arc features. Considering 5,670 sets of project-agnostic labels, three classifier types, and ten folds, `FlakeFriendEvaluator` produces 170,100 pairs of classifiers for S_1 and S_2 per project.

Table 6.3: The number of test cases, NOD and NODSpec flaky tests, and the total feature costs for the five feature-collecting modes. A dash (-) indicates the value is zero. An up tack (\perp) indicates there is no value because the run timed out after 72 hours.

Project	Tests			Total Feature Cost (seconds)				
	All	NOD	NODSpec	Static	Dynamic	DFunc	DLine	DFL
airflow	4116	18	17	1.37×10^2	4.79×10^3	8.78×10^3	6.63×10^4	7.15×10^4
core	21693	3	-	6.36×10^2	3.26×10^3	3.28×10^3	\perp	\perp
dask	10555	35	17	2.03×10^1	1.42×10^3	5.89×10^3	6.64×10^4	7.27×10^4
fonttools	2039	-	-	1.14×10^1	9.40×10^1	1.88×10^2	1.15×10^3	1.25×10^3
hypothesis	5429	5	-	9.47×10^0	1.84×10^3	7.33×10^3	6.51×10^3	1.42×10^4
ipython	865	3	3	4.44×10^0	7.25×10^1	1.16×10^2	2.27×10^3	2.27×10^3
prefect	7200	18	15	1.75×10^1	9.50×10^2	1.69×10^3	1.94×10^4	2.00×10^4
salt	4703	22	11	3.14×10^1	1.83×10^3	2.62×10^3	2.19×10^5	2.23×10^5
urllib3	1516	17	12	2.19×10^0	9.86×10^2	1.02×10^3	1.91×10^3	1.94×10^3
xonsh	4974	14	9	4.30×10^0	2.40×10^2	3.57×10^2	3.08×10^4	3.09×10^4
Total	63090	135	84	8.74×10^2	1.55×10^4	3.13×10^4	4.14×10^5	4.38×10^5

6.3.5 Project-Specific Prediction

For each set of project-agnostic labels from the Project-Agnostic Oracle stage and for each classifier type, `FlakeFriendEvaluator` takes the ten pairs of trained classifiers for S_1 and S_2 from the previous stage, corresponding to the ten folds, and applies both classifiers to the test cases of the testing portion of each fold. This results in two project-specific labels s_i^1 and s_i^2 for every test case t_i in the subject project associated with the set of project-agnostic labels. To ensure the labelling restriction holds, `FlakeFriendEvaluator` does not apply S_2 to any test case t_i where $s_i^1 = 0$ and will assume $s_i^2 = 0$. The output at this stage is 17,100 sets of project-specific labels per project.

6.4 Evaluation

Using the implementation of `FLAKEFRIEND` and 10 open-source Python projects as subjects, this empirical evaluation addressing the following research questions:

RQ1. What is the detection performance and time cost of `FLAKEFRIEND` for existing test cases?

RQ2. What is the detection performance of `FLAKEFRIEND` for future test cases?

RQ3. To what extent does `FLAKEFRIEND` reduce cumulative time cost as the test suite grows?

6.4.1 Subjects

Table 6.2 lists the ten subject projects in this evaluation. Each row shows the name of the GitHub repository, the shortened SHA of the commit, and the topic classifiers declared by the project’s maintainers. With the exception of `home-assistant/core`, these projects are a subset of those from the evaluation of `CANNIER` (see Table 5.3). Like the others, `home-assistant/core` is considered one of the most critical Python projects to open-source infrastructure [260] and is also one of the largest Python projects hosted on GitHub.

6.4.2 RQ1. What is the detection performance and time cost of `FLAKEFRIEND` for existing test cases?

`FlakeFriendEvaluator` produces thousands of sets of project-specific labels for every subject and compares them to their corresponding ground-truth labels to assess the detection performance for S_1 and S_2 . It calculates the confusion matrix (true-positive, true-negative, false-positive, false-negative) and the Matthews Correlation Coefficient (MCC) [24]. For each set of project-specific labels, `FlakeFriendEvaluator` calculates these metrics for A_1 and A_2 as well, using the associated

Table 6.4: The classifier types for the project-agnostic (**Agn.**) and project-specific (**Spe.**) models, the `pytest-FlakeFriend` mode, and the lower- and upper-thresholds at the knee and maximum points. In addition, the classifier type, mode, and threshold representative of classifier-only detection for RQ2.

Project	Knee					Maximum					Classifier		
	Type		Mode	ω_l	ω_u	Type		Mode	ω_l	ω_u	Type	Mode	ω_l/ω_u
	Agn.	Spe.				Agn.	Spe.						
airflow	E+K	ET	DLine	0.16	0.72	ET	ET	DFunc	0.08	0.84	ET	DFunc	0.08
core	ET	KNN	DFunc	0.00	0.20	ET	KNN	Dynamic	0.04	0.28	ET	Dynamic	0.12
dask	ET	KNN	Static	0.08	0.68	ET	ET	DLine	0.00	0.56	ET	Static	0.08
hypothesis	ET	ET	Dynamic	0.12	0.16	ET	E+K	DFunc	0.00	0.08	ET	Dynamic	0.12
ipython	E+K	KNN	Dynamic	0.08	0.08	E+K	KNN	Dynamic	0.08	0.08	ET	DFunc	0.28
prefect	E+K	KNN	DFunc	0.00	0.08	KNN	E+K	DLine	0.00	0.60	ET	DFL	0.36
salt	ET	ET	Dynamic	0.16	0.52	ET	ET	Dynamic	0.04	0.52	E+K	DFunc	0.12
urllib3	E+K	ET	DFunc	0.00	0.04	E+K	ET	DFL	0.00	0.76	ET	DFunc	0.16
xonsh	ET	KNN	DLine	0.08	1.04	ET	KNN	DLine	0.08	1.04	ET	DLine	0.08

set of project-agnostic labels. It also calculates the time cost of acquiring the project-agnostic labels as the sum of the total feature cost and the total oracle cost.

Given 17,100 sets of project-specific labels per subject, I cannot compare them all. Considering them as points in a two-dimensional space over the MCC for S_1 and the time cost, each associated with a classifier type for the project-agnostic and project-specific models, a mode, and threshold values, `FlakeFriendEvaluator` identifies two points of interest per subject. The first is the *knee point* on the Pareto front for maximizing MCC and minimizing time cost — the point with the shortest Euclidean distance to the *utopia point* [170], the theoretical point with an MCC of 1 and a time cost of 0. The second is the *maximum point*, with the locally-minimum time cost among those with the globally-maximum MCC.

To address this question, I compare the time cost and the performance metrics for A_1 and A_2 between the points of interest for each subject. As a baseline, `FlakeFriendEvaluator` calculates the time cost of acquiring the ground-truth labels via exhaustive rerunning. In this context, that is the time cost of executing the test suite 10,000 times as described in Section 6.3.2.

6.4.3 RQ2. What is the detection performance of FLAKEFRIEND for future test cases?

I compare the performance metrics for S_1 and S_2 , as before. `FlakeFriendEvaluator` calculates the metrics for detecting flaky tests using just a classifier as a baseline. For each project, it calculates these from the set of project-agnostic labels associated with the set of project-specific labels associated with the greatest MCC for A_1 with equal thresholds ($\omega_l = \omega_u$). Equal thresholds means the Project-Agnostic Oracle never resorts to rerunning, which is equivalent to using a classifier on its own with a single threshold on the predicted probability at which to declare a test case flaky.

6.4.4 RQ3. To what extent does FLAKEFRIEND reduce cumulative time cost as the test suite grows?

I compare the cumulative time cost of applying FLAKEFRIEND normally as the subjects' test suites grow in size to that of applying the Project-Agnostic Oracle stage to future test cases instead of the Project-Specific Prediction stage. In Figure 6.1, this would correspond to sending future test cases down the same pipeline as the existing test cases and stopping after the Project-Agnostic Oracle stage, producing project-agnostic labels for the future test cases instead of project-specific labels. This is representative of applying the CANNIER approach on its own (see Section 5.3). `FlakeFriendEvaluator` calculates the initial time cost of applying both approaches to each subject

using the parameters at both points of interest. This is the cost of producing feature vectors and project-agnostic labels for every existing test case, $\sum_{i=1}^{k-1}(\alpha_i + \beta_i)$. The cost of acquiring a project-specific label for a future test case t_i is α_i and the cost of acquiring a project-agnostic label is $\alpha_i + \beta_i$. As the expected values for these time costs for an arbitrary future test case, **FlakeFriendEvaluator** uses the mean value between all the existing test cases. It is able to calculate the expected cumulative time cost of acquiring project-agnostic labels for the existing test cases and project-specific labels for the future test cases, modelling **FLAKEFRIEND**, as:

$$\sum_{i=1}^{k-1}(\alpha_i + \beta_i) + (n - k + 1) \frac{\sum_{i=1}^{k-1} \alpha_i}{k - 1} \quad (6.3)$$

For the expected cumulative time cost of acquiring project-agnostic labels for all test cases, modelling **CANNIER**:

$$\sum_{i=1}^{k-1}(\alpha_i + \beta_i) + (n - k + 1) \frac{\sum_{i=1}^{k-1}(\alpha_i + \beta_i)}{k - 1} \quad (6.4)$$

6.4.5 Threats to Validity

FlakeFriendEvaluator may label some flaky tests as non-flaky in the ground-truth labels, since they could have exhibited flakiness only after more reruns. However, 10,000 is among the highest in the literature and the relationship with the number of detected flaky tests is sublinear [8]. The conclusions of this study would be the same had **FlakeFriendEvaluator** performed more, since it is unlikely that it would have detected many more flaky tests.

FlakeFriendEvaluator uses a Mann-Whitney U test with a 95% interval when deciding if a flaky test is **NODSpec** flaky. One would expect around 5% of those labelled as such to be false-positives. Using a more conservative interval results in far fewer positive examples, meaning training classifiers for A_2 and S_2 becomes impossible. Therefore, accepting that a small number of false-positives may exist in the ground-truth labels is necessary for the empirical evaluation. In general, when studying a non-deterministic phenomenon such as flaky tests, some level of uncertainty in the results is unavoidable.

The results of this study might not generalise to other Python projects. The subjects are in the top-200 most critical to open-source infrastructure [260]. While this does not guarantee generalisability, it indicates that the flaky tests in these projects could cause a more serious problem compared to less critical projects. The results may also not generalise to projects written in other languages. However, since **FLAKEFRIEND** does not make use of any Python-specific features, there is no reason it could not be implemented in other languages.

The implementation may contain bugs that affect the results of the evaluation. To mitigate this, I used well-established Python libraries such as **Coverage.py** [211], **psutil** [264], **Radon** [279], and **scikit-learn** [271]. These are highly active open-source projects and the chance that any serious bugs would be identified and fixed in a timely manner is high.

6.5 Results

For each subject, Table 6.3 shows the number of test cases, the number labelled by **FlakeFriendEvaluator** as **NOD** flaky, the number labelled as **NODSpec**, and the total feature costs for each of the five feature-collecting **pytest-FlakeFriend** modes. For **core**, the values for **DLine** and **DFL** are missing because the test suite run timed out after 72 hours. Table 6.4 shows the classifier types for the project-agnostic and project-specific models, the mode, and the lower- and upper-thresholds at the knee and maximum points for each subject except for **fonttools**, since it has no flaky tests. It also shows the classifier type, mode, and threshold representative of classifier-only detection for **RQ2**. Some subjects contain so few flaky tests that their **MCC** is unreliable. These projects would benefit very little from **FLAKEFRIEND**, and so focusing on those with more makes for a fairer assessment. Therefore, I disregard subjects with fewer than ten **NOD** flaky tests when calculating statistics.

Table 6.5: The performance metrics for detecting existing NOD and NODSpec flaky tests with FLAKE-FRIEND at the knee and maximum points. A dash (-) indicates the value is exactly zero. An up tack (\perp) indicates there is no value due to a division by zero.

Project	Knee					Maximum				
	TP	TN	FP	FN	MCC	TP	TN	FP	FN	MCC
NOD										
airflow	8	4093	5	10	0.52	16	4098	-	2	0.94
core	3	21670	20	-	0.36	3	21680	10	-	0.48
dask	17	10518	2	18	0.66	35	10517	3	-	0.96
hypothesis	4	5206	218	1	0.12	5	4846	578	-	0.09
ipython	3	848	14	-	0.42	3	848	14	-	0.42
prefect	18	6875	307	-	0.23	18	7180	2	-	0.95
salt	12	4680	1	10	0.71	20	4680	1	2	0.93
urllib3	17	1417	82	-	0.40	17	1499	-	-	1.00
xonsh	12	4960	-	2	0.93	12	4960	-	2	0.93
NODSpec										
airflow	7	4096	3	10	0.54	15	4099	-	2	0.94
core	-	21693	-	-	\perp	-	21693	-	-	\perp
dask	15	10538	-	2	0.94	17	10538	-	-	1.00
hypothesis	-	5429	-	-	\perp	-	5429	-	-	\perp
ipython	-	862	-	3	\perp	-	862	-	3	\perp
prefect	8	7185	-	7	0.73	15	7183	2	-	0.94
salt	7	4692	-	4	0.80	11	4692	-	-	1.00
urllib3	10	1504	-	2	0.91	12	1504	-	-	1.00
xonsh	8	4965	-	1	0.94	8	4965	-	1	0.94

6.5.1 RQ1. What is the detection performance and time cost of FLAKE-FRIEND for existing test cases?

Table 6.5 shows the performance metrics for detecting existing NOD and NODSpec flaky tests with FLAKE-FRIEND at the knee and maximum points. Table 6.6 shows the time cost at both points, compared to the time cost of exhaustive rerunning. When detecting NOD flaky tests with FLAKE-FRIEND at the knee point, the mean MCC is 0.57 and the standard deviation is 0.22. When detecting NODSpec flaky tests, the MCC is considerably higher and more consistent, with a mean of 0.81 and a standard deviation of 0.15. The mean time cost at the knee point is 2.15×10^6 seconds. When detecting NOD flaky tests at the maximum point, the predictions are near-perfect, with a mean MCC of 0.95 and a standard deviation of 0.02. For NODSpec flaky tests, these figures are only marginally different, with a mean of 0.97 and a standard deviation of 0.03. However, the mean time cost at the maximum point is 1.03×10^7 seconds, which is 9.94 times that at the knee point on average. As a baseline, the mean time cost of rerunning is 1.14×10^7 seconds. This is 10.7 and 1.62 times that of FLAKE-FRIEND at the knee and maximum points respectively.

Conclusion for RQ1. When detecting existing NOD and NODSpec flaky tests with FLAKE-FRIEND at the knee point, the mean MCC is 0.57 and 0.81 respectively. At the maximum point, the mean MCC is 0.95 and 0.97 respectively, but the time cost is 9.94 times that at the knee point on average. The mean time cost of exhaustive rerunning is 1.14×10^7 seconds. This is 10.7 and 1.62 times that of FLAKE-FRIEND at the knee and maximum points respectively.

Table 6.6: The time cost, in seconds, of detecting existing flaky tests with FLAKEFRIEND at the knee and maximum points, compared to exhaustive rerunning.

Project	FLAKEFRIEND		
	Knee	Maximum	Rerunning
airflow	6.80×10^6	1.97×10^7	2.47×10^7
core	1.11×10^6	1.92×10^6	2.11×10^6
dask	2.25×10^6	2.30×10^7	2.32×10^7
hypothesis	6.16×10^7	2.54×10^8	3.46×10^8
ipython	7.25×10^1	7.25×10^1	3.72×10^5
prefect	1.88×10^6	7.12×10^6	7.20×10^6
salt	1.64×10^6	4.76×10^6	5.61×10^6
urllib3	1.76×10^5	6.82×10^6	6.82×10^6
xonsh	1.76×10^5	1.76×10^5	7.57×10^5

6.5.2 RQ2. What is the detection performance of FLAKEFRIEND for future test cases?

Table 6.7 shows the metrics for detecting future NOD and NODSpec flaky tests with FLAKEFRIEND at the knee and maximum points, compared to with just a classifier. When detecting NOD flaky tests with FLAKEFRIEND at the knee point, the mean MCC is 0.52 and the standard deviation is 0.24. The detection performance is better and more consistent at the maximum point, where the mean MCC is 0.73 and the standard deviation is 0.09. The performance of the classifier-only approach is consistently only marginally better than random guessing, with a mean MCC of just 0.18 and a standard deviation of 0.10. For NODSpec flaky tests, the majority of the values in Table 6.7 are missing. This is because `FlakeFriendEvaluator` skips the Project-Specific Training stage if there are fewer than ten training examples of either class before balancing, due to the requirements of my chosen implementation of the classifiers. Therefore, I cannot draw conclusions about detecting future NODSpec flaky tests.

Conclusion for RQ2. When detecting future NOD flaky tests with FLAKEFRIEND at the knee point, the mean MCC is 0.52. At the maximum point, the mean MCC is 0.73. The detection performance of the classifier-only approach is consistently poor, with a mean MCC of just 0.18.

6.5.3 RQ3. To what extent does FLAKEFRIEND reduce cumulative time cost as the test suite grows?

Table 6.8 shows the expected cumulative time cost of FLAKEFRIEND and CANNIER at the knee and maximum points after extending the test suite by 1,000 test cases and to twice its original size. At the knee point, FLAKEFRIEND saves an average of 4.62×10^5 seconds in cumulative time cost after introducing 1,000 test cases compared to CANNIER. After doubling the size of the test suite, FLAKEFRIEND saves 2.14×10^6 seconds on average. The savings are greater at the maximum point, where FLAKEFRIEND saves 2.30×10^6 seconds after adding 1,000 test cases to the test suite and 1.02×10^7 seconds after extending it to twice its size.

Conclusion for RQ3. At the knee point, FLAKEFRIEND saves 4.62×10^5 seconds in cumulative time cost after introducing 1,000 test cases and 2.14×10^6 after doubling the size of the test suite. The savings at the maximum point are 2.30×10^6 and 1.02×10^7 seconds respectively.

Table 6.7: The performance metrics for detecting future NOD and NODSpec flaky tests with FLAKEFRIEND at the knee and maximum points, compared to with just a classifier. A dash (-) indicates the value is exactly zero. An up tack (\perp) indicates there is no value due to a division by zero or having fewer than ten training examples of either class during the Project-Specific Training stage.

Project	FLAKEFRIEND														
	Knee					Maximum					Classifier				
	TP	TN	FP	FN	MCC	TP	TN	FP	FN	MCC	TP	TN	FP	FN	MCC
NOD															
airflow	5	4097	1	13	0.48	11	4095	3	7	0.69	16	2923	1175	2	0.09
core	2	21678	12	1	0.31	2	21686	4	1	0.47	1	20916	774	2	0.02
dask	15	10520	-	20	0.65	23	10516	4	12	0.75	17	9751	769	18	0.09
hypothesis	3	5181	243	2	0.08	5	4896	528	-	0.09	4	4860	564	1	0.07
ipython	3	859	3	-	0.71	3	859	3	-	0.71	3	859	3	-	0.71
prefect	17	6877	305	1	0.22	12	7178	4	6	0.71	1	7182	-	17	0.24
salt	12	4680	1	10	0.71	17	4678	3	5	0.81	15	4503	178	7	0.22
urllib3	10	1405	94	7	0.22	7	1497	2	10	0.56	2	1470	29	15	0.07
xonsh	11	4959	1	3	0.85	11	4959	1	3	0.85	12	4887	73	2	0.34
NODSpec															
airflow	\perp	\perp	\perp	\perp	\perp	11	4096	3	6	0.71	-	4099	-	17	\perp
core	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	-	21693	-	-	\perp
dask	15	10538	-	2	0.94	15	10538	-	2	0.94	-	10504	34	17	0.00
hypothesis	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	-	5429	-	-	\perp
ipython	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	-	862	-	3	\perp
prefect	\perp	\perp	\perp	\perp	\perp	7	7184	1	8	0.64	-	7185	-	15	\perp
salt	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	-	4692	-	11	\perp
urllib3	\perp	\perp	\perp	\perp	\perp	5	1503	1	7	0.59	-	1504	-	12	\perp
xonsh	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	-	4965	-	9	\perp

6.6 Discussion

6.6.1 Detection Performance

The MCC when detecting existing and future flaky tests with FLAKEFRIEND at the maximum point is typically greater than that at the knee point. This is to be expected, given how `FlakeFriendEvaluator` selects the maximum point. Looking at Table 6.4, the greater detection performance is most likely caused by the wider separation of the thresholds at the maximum point. In practice, this would mean that the Project-Agnostic Oracle stage would delegate more existing test cases to exhaustive rerunning for their project-agnostic label. In the context of this evaluation, `FlakeFriendEvaluator` would use the value of the ground-truth labels as the project-agnostic labels for more test cases. Since it calculates the performance metrics with respect to the ground-truth labels, this results in a greater MCC. Furthermore, better project-agnostic labels are more likely to result in better project-specific models and therefore better project-specific labels, and a greater MCC when detecting future flaky tests.

The mean MCC of FLAKEFRIEND when detecting existing NODSpec flaky tests at the knee point is 0.81, compared to 0.57 when detecting existing NOD flaky tests. This could be due to the labelling restriction imposed by the Project-Agnostic Oracle stage, namely $a_i^j = 1 \implies a_i^1 = 1$ for $j > 1$. This means that `FlakeFriendEvaluator` will not label a test case as NODSpec flaky if it did not previously label the test case as NOD flaky. At the knee point, FLAKEFRIEND has a mean positive rate of 0.02 and a mean recall of 0.72 when detecting NOD flaky tests. This means that, on average, `FlakeFriendEvaluator` can only give 2% of test cases a positive project-agnostic label for NODSpec flakiness, and this sample accounts for 72% of the genuine NOD flaky test cases.

Table 6.8: The expected cumulative time cost of FLAKEFRIEND and CANNIER at the knee and maximum points after extending the test suite by 1,000 test cases (+1,000) and to twice its original size ($\times 2$).

Project	Knee				Max			
	FLAKEFRIEND		CANNIER		FLAKEFRIEND		CANNIER	
	+1,000	$\times 2$	+1,000	$\times 2$	+1,000	$\times 2$	+1,000	$\times 2$
airflow	6.82×10^6	6.87×10^6	8.62×10^6	1.36×10^7	1.97×10^7	1.97×10^7	2.45×10^7	3.94×10^7
core	1.11×10^6	1.12×10^6	1.16×10^6	2.23×10^6	1.92×10^6	1.93×10^6	2.01×10^6	3.85×10^6
dask	2.25×10^6	2.25×10^6	2.46×10^6	4.50×10^6	2.30×10^7	2.31×10^7	2.53×10^7	4.61×10^7
hypothesis	6.16×10^7	6.16×10^7	7.30×10^7	1.23×10^8	2.54×10^8	2.54×10^8	3.01×10^8	5.07×10^8
ipython	1.57×10^2	1.45×10^2	1.57×10^2	1.45×10^2	1.57×10^2	1.45×10^2	1.57×10^2	1.45×10^2
prefect	1.88×10^6	1.89×10^6	2.15×10^6	3.77×10^6	7.12×10^6	7.14×10^6	8.25×10^6	1.42×10^7
salt	1.64×10^6	1.64×10^6	1.99×10^6	3.29×10^6	4.76×10^6	4.76×10^6	5.77×10^6	9.51×10^6
urllib3	1.77×10^5	1.77×10^5	2.92×10^5	3.52×10^5	6.82×10^6	6.83×10^6	1.14×10^7	1.36×10^7
xonsh	1.83×10^5	2.07×10^5	2.13×10^5	3.53×10^5	1.83×10^5	2.07×10^5	2.13×10^5	3.53×10^5

Because a test case cannot be NODSpec flaky if it is not NOD flaky, this means that A_2 has a significantly smaller margin for error than A_1 and thus a greater mean MCC.

The MCC of FLAKEFRIEND when detecting existing flaky tests is typically greater than that when detecting future flaky tests. For example, the mean MCC for NOD flaky tests at the maximum point is 0.95 for existing and 0.73 for future. This is unsurprising since `FlakeFriendEvaluator` calculates the former from the project-agnostic labels from A_1 , which forms the training data for S_1 , which produces the project-specific labels from which it calculates the latter. The advantage of the project-specific models is that they do not impose an oracle cost, and so the lower detection performance may be considered a trade-off in favour of a lower cumulative time cost.

6.6.2 Time Cost

The mean time cost of obtaining project-agnostic labels for existing test cases with FLAKEFRIEND at the knee point is 2.15×10^6 seconds. This may seem prohibitive, but it is a single-core time cost. Correct as of May 2023, Amazon offers a `c6g.8xlarge` instance, with 32 CPU cores and 64 GiB of memory, at the on-demand hourly rate of 1.088 USD [216]. On this instance, the mean time cost would be $((2.15 \times 10^6) \div 32) \div 60^2 \approx 18.7$ hours. This translates into a monetary cost of $18.7 \times 1.088 \approx 20.3$ USD. As an upper-bound to compare to, consider the mean time cost of exhaustive rerunning, 1.14×10^7 seconds. On a `c6g.8xlarge` instance, this would require 99 hours at a cost of 108 USD.

At the knee point, FLAKEFRIEND saves 4.62×10^5 seconds in cumulative time cost after introducing 1,000 test cases and 2.14×10^6 after extending it to twice its original size. The savings at the maximum point are 2.30×10^6 and 1.02×10^7 seconds respectively. This is compared to CANNIER. On a `c6g.8xlarge` instance at the knee point, this saving equates to 4.01 hours and 4.36 USD after adding 1,000 test cases and 18.6 hours and 20.2 USD after doubling the size of the test suite. At the maximum point, it equates to 20 hours and 21.7 USD and, respectively, 88.5 hours and 92.4 USD.

6.6.3 Implications for Developers

This evaluation has demonstrated that FLAKEFRIEND can address the drawbacks of both rerunning-based and machine learning-based flaky test detection. It has also shown that FLAKEFRIEND offers software developers benefits that CANNIER cannot by reducing the cumulative time cost of detecting and classifying flaky tests as the test suite evolves over time. Following an off-line training stage to produce project-specific models, developers could configure their continuous integration system to execute the Specific Prediction stage after commits that introduce new

test cases. This is beneficial because it makes developers aware of flaky tests immediately after introducing them, which means flaky tests will be less likely to accumulate [180]. The additional information that FLAKEFRIEND can offer with regards to the category of flakiness may also help developers repair flaky tests [56].

6.7 Related Work

Pinto et al. [128] evaluated five types of machine learning classifiers for detecting NOD flaky tests in Java projects, including K-Nearest Neighbors and Random Forest [18]. Like this study, they used sparse features encoding the occurrence of tokens in the test case code and a small number of static metrics. Unlike this study, they did not investigate any other types of features. In their empirical evaluation, they reported very good detection performance for all types of classifier. This contrasts with the results of this evaluation, where I found classifier-only detection to be generally very poor. However, Pinto et al. calculated detection performance metrics via cross-validation over the entire set of 64,382 test cases from their 12 subject projects. Therefore, their evaluation did not take into account inter-project generalisability, and projects with more test cases would have had a greater impact on their results.

Verdecchia et al. [159] performed a related study focusing on tokens in test case code and a K-Nearest Neighbors classifier with varying meta-parameters. They prefaced their classifiers with a stage of Random Projection to reduce data dimensionality, a setup that I also used in this study. Like Pinto et al., they evaluated their classifiers via cross-validation over the entire set of test cases. However, they calculated performance metrics with respect to each of their subject projects. While this prevents particular projects skewing the results, it still does not account for inter-project generalisability.

Alshammari et al. [8] evaluated seven types of classifier. They considered token-based features, but introduced additional dynamic metrics including the execution time and the number of covered lines. They found that adding these metrics led to a greater detection performance, at the cost of having to execute the test suite. In the study presented in Chapter 4, I found that extending this set of dynamic metrics with properties including the peak number of threads improved detection performance even further. I used many of the metrics from both studies in this evaluation (see Table 6.1). The methodologies of these two studies were similar to Verdecchia et al. in that they evaluated classifiers using cross-validation over the entire set of test cases but calculated per-project performance metrics.

Camara et al. [20] performed a replication study of Pinto et al.. They confirmed the poor inter-project generalisability of classifiers for detecting flaky tests. Following the same procedure as Pinto et al., but evaluating classifiers using cross-project validation as in this study, they observed markedly lower detection performance. Fatima et al. [41] observed a similar result in their evaluation of a classifier for flaky tests based on CodeBERT [43]. Akli et al. [6] evaluated machine learning classifiers based on CodeBERT for the prediction of flaky test categories. They manually labelled the ground-truth categories of the flaky tests in their dataset, whereas I used an automated approach (see Section 6.3.2). While predicting categories is a feature of FLAKEFRIEND, it is not its primary contribution, which is instead the reduced cumulative time cost over repeated use as the test suite evolves.

6.8 Conclusion

I presented FLAKEFRIEND for detecting and classifying existing flaky tests in a project and producing machine learning classifiers to provide predictions for future test cases in that project. The evaluation considered a specific category of flaky tests, NODSpec, that is non-order-dependent flaky tests that appear to be dependent on the CPU and/or disk read/write speed of the machine. I demonstrated that FLAKEFRIEND is able to detect and classify existing flaky tests in a project considerably faster than exhaustive rerunning. I found that the detection performance of the

classifiers produced by FLAKEFRIEND for a project, with respect to the future flaky tests in that project, is significantly greater than classifiers pre-trained on the test cases of other projects. I also found that FLAKEFRIEND offers considerable savings in cumulative time cost when detecting flaky tests in evolving test suites compared to CANNIER. These findings demonstrate that developers can save time in the long run by using FLAKEFRIEND to detect and classify flaky tests, making it suitable to be implemented as part of a continuous integration system, automatically triggered when developers introduce new test cases.

Chapter 7

Conclusion and Future Work

In this thesis, I enhanced the understanding of the manifestation, causes, and impacts of flaky tests. I also presented techniques to mitigate the problem of flaky tests by automatically detecting them, which I empirically evaluated on large datasets of test cases from open-source projects. See Table 7.1 for a summary of the primary research chapters of this thesis.

In Chapter 2, I systematically reviewed 76 published papers on the topic. In doing so, I identified a taxonomy of causes of flaky tests, and found that they impose negative effects on the reliability and efficiency of testing in general, as well as being a hindrance to a host of techniques in software engineering. I also presented a range of techniques for mitigating some of the problems presented by flaky tests and described a technique for the automatic repair of order-dependent tests. In Chapter 3, I deployed an online survey about flaky tests, not restricted to any particular organisation, and received 170 responses. I also procured a dataset of 38 StackOverflow threads, upon which I performed thematic analysis. The analyses of both sources focused on how developers define and react to flaky tests and their experiences of the causes and impacts. From the findings, I was able to offer six actionable recommendations for both researchers and developers.

In Chapter 4, I presented FLAKE16, a new feature set that encodes test cases for machine-learning-based flaky test detection. I evaluated the performance of 54 machine learning pipelines when detecting both non-order-dependent and order-dependent flaky tests using both FLAKE16 and a previously established feature set [8]. For both categories, experiments involving 26 real-world Python projects showed greater detection performance when using FLAKE16. In Chapter 5, I presented CANNIER, an approach for reducing the time cost of rerunning-based detection techniques by combining them with machine learning classifiers. Following an evaluation involving 30 Python projects, I found that CANNIER was able to reduce the time cost of three existing rerunning-based methods for flaky test detection, including IDFLAKIES [90], by an order of magnitude at the expense of only a minor decrease in detection performance. In Chapter 6, I presented FLAKEFRIEND for detecting and classifying existing flaky tests in a project and producing machine learning classifiers to provide predictions for future test cases in that project. I demonstrated that FLAKEFRIEND is able to detect and classify existing flaky tests in a project considerably faster than exhaustive rerunning. Furthermore, I found that the detection performance of the classifiers produced by FLAKEFRIEND for a project, with respect to the future flaky tests in that project, is significantly greater than classifiers pre-trained on the test cases of other projects.

7.1 Restrictions and Limitations

7.1.1 Types of Subject Programs

The empirical evaluations throughout this thesis focus on open-source subject programs written in the Python programming language. This means that I cannot generalise the findings to proprietary programs or those written in other languages with complete certainty. Conducting open research

Table 7.1: Summary of the primary research chapters of this thesis.

#	Summary
3	<p>Technique: -</p> <p>Subjects: 170 developer participants, 38 StackOverflow threads.</p> <p>Findings: 1. Some developers suggest that the definition of a flaky test should extend beyond the test case code or the code under test. 2. Developers strongly agree that flaky tests hinder continuous integration. 3. Developers rated improper setup and teardown as the most common cause of flaky tests. 4. Developers rated rerunning the failing build as the most common action they take when encountering flaky tests.</p>
4	<p>Technique: FLAKE16</p> <p>Subjects: 67,006 test cases from 26 open-source Python projects.</p> <p>Findings: 1. FLAKE16 offered a 13% increase in overall F1 score when detecting NOD flaky tests and a 17% increase when detecting OD flaky tests compared to FLAKEFLAGGER. 2. Machine learning classifiers are just as applicable to detecting OD flaky tests as they are to detecting NOD flaky tests. 3. The most impactful feature when detecting NOD flaky tests was the peak number of concurrently running threads. When detecting OD flaky tests, the number of read- and write-related system calls was the most impactful.</p>
5	<p>Technique: CANNIER</p> <p>Subjects: 89,668 test cases from 30 open-source Python projects.</p> <p>Findings: 1. The performance of machine learning classifiers alone for flaky test detection is lacklustre and variable between projects. 2. The relationship between the number of repeated feature measurements to produce mean feature vectors and the overall MCC of a classifier is positive. 3. Some test case metrics have a clear effect on the output of a classifier for detecting flaky tests and others appear to have more complex relationships. 4. CANNIER is able to reduce time cost by an average of 88% between three previously established rerunning-based flaky test detection techniques.</p>
4	<p>Technique: FLAKEFRIEND</p> <p>Subjects: 63,090 test cases from 10 open-source Python projects.</p> <p>Findings: 1. When detecting existing NOD flaky tests with FLAKEFRIEND at the knee and maximum points, the mean MCC is 0.57 and 0.95 respectively. The mean time cost of exhaustive rerunning is 10.7 and 1.62 times that of FLAKEFRIEND at the two points respectively. 2. When detecting future NOD flaky tests with FLAKEFRIEND at the knee and maximum points, the mean MCC is 0.52 and 0.72 respectively. The mean MCC of the classifier-only approach is 0.18. 3. At the knee point, FLAKEFRIEND saves 4.62×10^5 seconds in cumulative time cost after introducing 1,000 test cases and 2.14×10^6 after doubling the size of the test suite. The savings at the maximum point are 2.30×10^6 and 1.02×10^7 seconds respectively.</p>

involving proprietary software can present unique challenges, such as restrictions imposed by non-disclosure agreements. However, this is not unprecedented in the literature [88]. The techniques presented in this thesis are broadly language-agnostic, in the sense that they do not rely on Python-specific behaviour. This means they could be implemented in other languages.

7.1.2 Empirical Uncertainty

Many results in this thesis stem from empirical evaluations involving datasets of flaky tests. I produced these datasets myself by repeatedly executing the test suites of open-source projects in various ways. As shown in Figure 4.1, and found by other studies [8], one is likely to continue to detect flaky tests even after a significant number of reruns (e.g., 10,000). This implies that there may be flaky tests labelled as non-flaky in the datasets of this thesis, simply because more reruns would have been needed to detect them. This has the potential to affect the results. I performed as many reruns as was feasible given the time constraints and available computational resources.

7.2 Future Work

7.2.1 Extended Replication

I could replicate the empirical evaluations in this thesis with a wider subject set involving proprietary programs and programs written in languages other than Python. If the results were broadly similar, it would demonstrate that the findings of this thesis are indeed generalisable. If not, it would raise the interesting question of why not, which could warrant further investigation into the properties of software that make the techniques presented in this thesis more or less applicable. Such a replication presents various challenges, including establishing industrial partners willing to share their internal software for evaluation and reimplementing the techniques in other languages.

7.2.2 Extended Developer Survey

I could further build upon the understanding of flaky tests by extending the scope of Chapter 3. I could do this by conducting focused surveys in different types of communities. For instance, I could consider administering the survey at one or more of the following: a large company, a medium-sized company, a startup company, a government organisation, and an open-source project. I could also conduct open-ended follow-on video interviews with a small sample of respondents, leading to further insights beyond the questions I already asked. I could also analyze new sources of external data beyond the StackOverflow threads, including GitHub Discussions and the GitHub issue tracker. While flaky tests may not be frequently discussed in such channels, there might be more discussions about flaky tests before merging a pull request. Finally, I could interview the founders and engineers at startup companies that build tools for handling flaky tests, such as BuildPulse [207], as they may be willing to characterise the flaky tests their tool finds.

7.2.3 Automatically Generated Flakiness

Previous studies have established that automatic test case generation tools, such as EVOSUITE [44], can sometimes generate flaky tests [40, 125, 138]. In this thesis, and in many other studies involving machine learning-based flaky test detection, there is a persistent problem of highly imbalanced training data. Specifically, training sets contain many more examples of non-flaky tests than flaky tests. This has previously been addressed with interpolation techniques such as SMOTE [23], but it may be worth investigating if automated test generation tools can be used as a means of seeding training data with additional examples of flaky tests. While such examples would be synthetic, they would arguably be more realistic than the data points generated by SMOTE, which do not even correspond to real test cases.

7.2.4 Latent Flakiness

An order-dependent flaky test is called *latent* if it is not flaky in the current state of the test suite, but could become so in the future [162]. This is typically caused by a test case containing one or more brittle assertions that have the potential to be influenced by some shared state. If developers eventually introduce a new test case that pollutes that state, then the latent flaky test will be manifested. Arguably the best time to discover latent test flakiness is immediately after the developer has created the test case. I could facilitate this with an automated technique that attempts to generate a test case that induces flakiness in a given developer-written test case. The generated test case would demonstrate to the developer why their test case is flaky and serve as a guide to repair it. Such a technique could follow a search-based approach, attempting to generate a test case that when executed prior to the developer-written test case, induces it to execute in an unexpected fashion. The search could be driven by three fitness functions: the first to maximise the number of modifying operations performed by the generated test case upon mutable objects referenced by the developer-written test case; the second to consider how “close” assertions in the developer-written test case are to being evaluated differently; and the third to consider how “close” the path through the code under test is to being different from the default [119].

7.2.5 Automated Explanation

Leveraging the techniques I have already developed, I could develop a more comprehensive automated technique for mitigating flaky tests that not only detects them but also identifies the context leading to the flaky failure, from which to generate human-readable explanations. This would be of considerable value to developers, since an explanation of why a test case could be flaky is far more useful when attempting to repair it than simply flagging it as potentially flaky [56]. Such a technique would need to address three primary objectives. Firstly, it would need to automatically reproduce environmental conditions that occur during test case execution. My work in Chapter 6 involving the CPU and disk speed is a first step towards this goal, but I would need to investigate other conditions that could impact the outcome of a test case. Secondly, it would need to reliably reproduce the conditions that lead to passing and failing executions of a flaky test. This could be realised as a search-based technique where the environmental conditions, and the types of conditions, are the search space [109]. A major challenge is the development of a fitness function, since not all instances of test flakiness are feasible to control. Thirdly, from the conditions generated to induce passing and failing, it would need to generate a human-readable explanation. A key challenge here is minimizing the set of conditions to ensure the explanations only include relevant information and are actionable by developers.

References

- [1] H. Abdi and L. J Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.
- [2] A. Ahmad, F. de Oliveira Neto, Z. Shi, K. Sandahl, and O. Leifler. A multi-factor approach for flaky test detection and automated root cause analysis. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 338–348, 2021.
- [3] A. Ahmad, E. N. Held, O. Leifler, and K. Sandahl. Identifying randomness related flaky tests through divergence and execution tracing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 293–300, 2022.
- [4] A. Ahmad, O. Leifler, and K. Sandahl. An evaluation of machine learning methods for predicting flaky tests. In *Proceedings of the International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, pages 37–46, 2020.
- [5] A. Ahmad, O. Leifler, and K. Sandahl. Empirical analysis of practitioners’ perceptions of test flakiness factors. *Software Testing Verification and Reliability*, 31(8):1–24, 2021.
- [6] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. L. Traon. Predicting flaky tests categories using few-shot learning. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 293–300, 2023.
- [7] R. Al-Qutaish and A. Abran. *Halstead Metrics: Analysis of their Design*, pages 145–159. Wiley, 2010.
- [8] A. Alshammari, C. Morris, M. Hilton, and J. Bell. FlakeFlagger: Predicting flakiness without rerunning tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.
- [9] A. Alsharif, G. M. Kapfhammer, and P. McMinn. STICCER: Fast and effective database test suite reduction through merging of similar test cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2020.
- [10] A. Arcuri, G. Fraser, and J. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 79–89, 2014.
- [11] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *Explorations Newsletter*, 6(1):20–29, 2004.
- [12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.
- [13] J. Bell. Detecting, isolating, and enforcing dependencies among and within test cases. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 799–802, 2014.

- [14] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 550–561, 2014.
- [15] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 770–781, 2015.
- [16] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.
- [17] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella. Web test dependency detection. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 154–164, 2019.
- [18] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [19] B. Camara, M. Silva, A. Endo, and Vergilio S. On the use of test smells for prediction of flaky tests. In *Proceedings of the Brazilian Symposium on Systematic and Automated Software Testing (SAST)*, pages 46–54, 2021.
- [20] B. Camara, M. Silva, A. Endo, and Vergilio S. What is the vocabulary of flaky tests? An extended replication. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 444–454, 2021.
- [21] J. Campos, R. Abreu, G. Fraser, and M. d’Amorim. Entropy-based test generation for improved fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 257–267, 2013.
- [22] J. Candido, L. Melo, and M. D’Amorim. Test suite parallelization in open-source projects: A study on its usage and impact. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 153–158, 2017.
- [23] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.
- [24] G. Chicco, D. Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(6):1471–2164, 2020.
- [25] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. Localizing SQL faults in database applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2011.
- [26] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman. FlakiMe: Laboratory-Controlled test flakiness impact assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 982–994, 2022.
- [27] K. Costa, R. Ferreira, G. Pinto, M. D’Amorim, and B. Miranda. Test flakiness across programming languages. *Journal of Logic and Computation*, pages 1–14, 2022.
- [28] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
- [29] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011.

- [30] J. Dietrich, S. Rasheed, and A. Tahir. Flaky test sanitisation via on-the-fly assumption inference for tests with network dependencies. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 264–275, 2022.
- [31] S. Donaldson and E. Grant-Vallone. Understanding self-report bias in organizational behavior research. *Journal of Business and Psychology*, 17(2):245–260, 2002.
- [32] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury. Flaky test detection in android via event order exploration. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 367–378, 2021.
- [33] T. Durieux, C. L. Goues, M. Hilton, and R. Abreu. Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 254–264, 2020.
- [34] S. Dutta, A. Arunachalam, and S. Misailovic. To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 151–161, 2022.
- [35] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and Misailovic S. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 211–224, 2020.
- [36] S. Dutta, A. Shi, and Misailovic S. FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 211–224, 2021.
- [37] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer’s perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 830–840, 2019.
- [38] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15:485–499, 2003.
- [39] A. T. Endo and A. Moller. NodeRacer: Event race detection for Node.js applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 120–130, 2020.
- [40] Z. Fan. A systematic evaluation of problematic tests generated by EvoSuite. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 165–167, 2019.
- [41] S. Fatima, T. Ghaleb, and L. Briand. Flakify: A black-box, language model-based predictor for flaky tests. *Transactions on Software Engineering*, pages 1–17, 2022.
- [42] M. Fazzini, A. Gorla, and A. Orso. A framework for automated test mocking of mobile apps. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 1204–1208, 2020.
- [43] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M Gong, L. Shou, B. Qin, T. Liu, D. Jinag, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. *arXiv*, 2020.
- [44] G. Fraser and A. Arcuri. A large scale evaluation of automated unit test generation using EvoSuite. *Transactions on Software Engineering and Methodology*, 24:8, 2014.

- [45] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.
- [46] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.
- [47] A. Gambi, J. Bell, and A. Zeller. Practical test dependency detection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11, 2018.
- [48] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 55–65, 2015.
- [49] Z. Gao and A. Memon. Which of my failures are real? Using relevance ranking to raise true failures to the top. In *Proceedings of the International Conference on Automated Software Engineering Workshops (ASEW)*, pages 62–60, 2015.
- [50] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.
- [51] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.
- [52] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *Transactions on Software Engineering*, 17(12):1284, 1991.
- [53] S. W. Golomb and H. Taylor. Tuscan squares - a new family of combinatorial designs. *Ars Combinatoria*, 20:115–132, 1985.
- [54] S. Grafberger, J. Stoyanovich, and S. Schelter. Lightweight inspection of data preprocessing in native machine learning pipelines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.
- [55] A. Groce and J. Holmes. Practical automatic lightweight nondeterminism and flaky test detection and debugging for Python. In *Proceedings of the International Conference on Software Quality, Reliability, and Security (QRS)*, pages 188–195, 2021.
- [56] M. Gruber and G. Fraser. A survey on how test flakiness affects developers and what support they need to address it. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2022.
- [57] M. Gruber and G. Fraser. Debugging flaky tests using spectrum-based fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2023.
- [58] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser. Practical flaky test prediction using common code evolution and test history data. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2023.
- [59] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. An empirical study of flaky tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2021.
- [60] A. Gyori, B. Lambeth, S. Khurshid, and D. Marinov. Exploring underdetermined specifications using Java PathFinder. *Software Engineering Notes*, 41:1–5, 2017.
- [61] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specification. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 223–233, 2015.

- [62] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA)*, pages 223–233, 2015.
- [63] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2022.
- [64] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, and Y. L. Cordy, M. Traon. What made this test flake? pinpointing classes responsible for test flakiness. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 352–363, 2022.
- [65] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon. A replication study on the usability of code vocabulary in predicting flaky tests. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021.
- [66] F. Haftmann, D. Kossmann, and E. Lo. Parallel execution of test runs for database application systems. In *Proceedings of the International Conference on Very Large Data Bases*, 2005.
- [67] Shifa E.Zehra Haidry and Tim Miller. Using dependency structures for prioritization of functional test suites. *Transactions on Software Engineering*, 39:258–275, 2013.
- [68] M. Harman and P. O’Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23, 2018.
- [69] N. Hashemi, A. Tahir, and S. Rasheed. An empirical study of flaky tests in JavaScript. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34, 2001.
- [70] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 53–63, 2018.
- [71] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 197–207, 2017.
- [72] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 426–437, 2016.
- [73] G. Hooker and L. Mentch. Please stop permuting features: An explanation and alternatives. *ArXiv*, 2019.
- [74] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 621–631, 2014.
- [75] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.
- [76] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

- [77] G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*. 2004.
- [78] G. M. Kapfhammer. Regression testing. In *The Encyclopedia of Software Engineering*. 2010.
- [79] G. M. Kapfhammer, McMinn P., and C. J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2013.
- [80] J. M. Keller, M. R. Gray, and J. A. Givens. A fuzzy k-nearest neighbor algorithm. *Transactions on Systems, Man, and Cybernetics*, 15(4):580–585, 1985.
- [81] S. Kensen, Steinhardt J., and P. Liang. FrAngel: Component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3, 2019.
- [82] T. King, D. Santiago, J. Phillips, and P. Clarke. Towards a bayesian network model for predicting flaky automated tests. In *Proceedings of the International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pages 100–107, 2018.
- [83] B. Korel, L. H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 214–223, 2002.
- [84] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon. Modeling and ranking flaky tests at Apple. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 110–119, 2020.
- [85] J. Kupiec. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, 6(3):225–242, 1992.
- [86] A. Labuschagne, L. Inozemtseva, and R. Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 821–830, 2017.
- [87] F. Lacoste. Killing the gatekeeper: Introducing a continuous integration system. In *Proceedings of the Agile Conference (AGILE)*, pages 387–392, 2009.
- [88] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–215, 2019.
- [89] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1471–1482, 2020.
- [90] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. IDFlakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322, 2019.
- [91] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie. Dependent-test-aware regression testing techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 298–311, 2020.
- [92] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *Proceedings of the International Conference on Software Reliability Engineering (ISSRE)*, pages 403–413, 2020.
- [93] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(11):202–231, 2020.

- [94] J. Lampel, S. Just, S. Apel, and A. Zeller. When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1381–1392, 2021.
- [95] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.
- [96] T. Leesatapornwongsa, X. Ren, and S. Nath. FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1509–1520, 2022.
- [97] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco. Assessing transition-based test selection algorithms at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 101–110, 2019.
- [98] C. Li, C. Zhu, and A. Wang, W. Shi. Repairing order-dependent flaky tests via test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1881–1892, 2022.
- [99] S. Li and A. Shi. Evolution-aware detection of order-dependent flaky tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 114–125, 2022.
- [100] C. T. Lin, K. W. Tang, and G. M. Kapfhammer. Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests. *Information and Software Technology*, 56(10):1322–1344, 2014.
- [101] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA)*, pages 31–42, 2019.
- [102] V. López, A. Fernández, S. García, V. Palade, and F. Herrera. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250:113–141, 2013.
- [103] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himelfarb, N. Bansal, and S. Lee. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.
- [104] Q. Luo, L. Eloussi, F. Hariri, and D. Marinov. Can we trust test outcomes? *Network*, 5(1), 2014.
- [105] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 643–653, 2014.
- [106] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019.
- [107] J. Malm, A. Causevic, B. Lisper, and S. Eldh. Automated analysis of flakiness-mitigating delays. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 81–84, 2020.

- [108] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.
- [109] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2), 2004.
- [110] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 233–242, 2017.
- [111] J. Morán, C. Augusto, A. Bertolino, C. De La Riva, and J. Tuya. Debugging flaky tests on web applications. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST)*, pages 454–461, 2019.
- [112] R. Mudduluru, J. Waataja, S. Millstein, and M. D. Ernst. Verifying determinism in sequential programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.
- [113] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE)*, pages 496–499, 2011.
- [114] A. Najafi, P. C. Rigby, and W. Shang. Bisecting commits and modeling commit risk during testing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 279–289, 2019.
- [115] S. M. Nasehi, J. Sillito, F. Maurer, and C Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012.
- [116] M. Nejadgholi and J. Yang. A study of oracle approximations in testing deep learning libraries. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 785–796, 2019.
- [117] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric. Debugging the performance of maven’s test isolation: Experience report. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 249–259, 2020.
- [118] F. Palomba and A. Zaidman. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering*, 24(11):2907–2946, 2019.
- [119] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Flake it ‘till you make it: Using automated repair to induce and fix latent test flakiness. In *Proceedings of the International Workshop on Automated Program Repair (APR)*, pages 11–12, 2020.
- [120] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. A survey of flaky tests. *Transactions on Software Engineering and Methodology*, 31(1):1–74, 2021.
- [121] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 93–104, 2022.
- [122] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 253–262, 2022.

- [123] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 160–164, 2022.
- [124] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(72), 2023.
- [125] S. Paydar and A. Azamnouri. An experimental study on flakiness and fragility of Randoop regression test suites. *Lecture Notes in Computer Science*, pages 111–126, 2019.
- [126] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *International Conference on Software Engineering (ICSE)*, pages 524–536, 2021.
- [127] Q. Peng, A. Shi, and L. Zhang. Empirically revisiting and enhancing IR-based test-case prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 324–336, 2020.
- [128] G. Pinto, B. Miranda, S. Dissanayake, M. D. Amorim, C. Treude, A. Bertolino, and M. D’amorim. What is the vocabulary of flaky tests? In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 492–502, 2020.
- [129] V. Pontillo, F. Palomba, and F. Ferrucci. Toward static test flakiness prediction: A feasibility study. In *Proceedings of the International Workshop on Machine Learning Techniques for Software Quality Evoluton*, pages 19–24, 2021.
- [130] V. Pontillo, F. Palomba, and F. Ferrucci. Static test flakiness prediction: How far can we go? *Empirical Software Engineering*, pages 325–327, 2022.
- [131] J. A. Prado Lima and S. R. Vergilio. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121, 2020.
- [132] K. Presler-Marshall, E. Horton, S. Heckman, and K. T. Stolee. Wait wait. No, tell me. Analyzing Selenium configuration effects on test flakiness. In *Proceedings of the International Workshop on Automation of Software Testing (AST)*, pages 7–13, 2019.
- [133] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, and X. Mao. PEELER: Learning to effectively predict flakiness without running tests. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 257–268, 2022.
- [134] M. T. Rahman and P. C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 857–862, 2018.
- [135] M. H. U. Rehman and P. C. Rigby. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1371–1380, 2021.
- [136] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. An empirical analysis of UI-based flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.
- [137] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

- [138] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 201–211, 2015.
- [139] A. Shi, J. Bell, and D. Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 296–306, 2019.
- [140] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, 2016.
- [141] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 545–555, 2019.
- [142] A. Shi, P. Zhao, and D. Marinov. Understanding and improving regression test selection in continuous integration. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2019.
- [143] T. Shi and S. Horvath. Unsupervised learning with random forest predictors. *Journal of Computational and Graphical Statistics*, 15(1):118–138, 2006.
- [144] D. Silva, L. Teixeira, and M. D’Amorim. Shake it! Detecting flaky tests caused by concurrency with Shaker. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311, 2020.
- [145] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 402–412, 2017.
- [146] A. Stahlbauerm, M. Kreis, and G. Fraser. Testing scratch programs automatically. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 165–175, 2019.
- [147] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark. Intermittently failing tests in the embedded systems domain. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 337–348, 2020.
- [148] Y. Sun, A. K. C. Wong, and M. S. Kamel. Classification of imbalanced data: A review. *International Journal of Pattern Recognition and Artificial Intelligence*, 23(4):687–719, 2009.
- [149] S. Tahvili, M. Ahlberg, E. Fornander, W. Afzal, M. Saadatmand, M. Bohlin, and M. Sarabi. Functional dependency detection for integration test cases. In *International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 207–214, 2018.
- [150] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, and M. Bohlin. Automated functional dependency detection between test cases using Doc2Vec and clustering. In *International Conference On Artificial Intelligence Testing (AITest)*, pages 19–26, 2019.
- [151] V. Terragni, P. Salza, and F. Ferrucci. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 69–72, 2020.
- [152] S. Thorve, C. Sreshtha, and N. Meng. An empirical study of flaky tests in Android apps. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538, 2018.

- [153] Ivan Tomek. Two modifications of CNN. *Transactions on Systems, Man, and Cybernetics*, 6:769–772, 1976.
- [154] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 4–15, 2016.
- [155] A. Vahabzadeh, A. A. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, 2015.
- [156] L. Van Der Maaten, E. Postma, and J. Van den Herik. Dimensionality reduction: A comparative. *Journal of Machine Learning Research*, 10(66-71):13, 2009.
- [157] B. Vancsics, T. Gergely, and A. Beszédes. Simulating the effect of test flakiness on fault localization effectiveness. In *Proceedings of the International Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 28–35, 2020.
- [158] B. Vaysburg, L. H. Tahat, and B. Korel. Dependence analysis in reduction of requirement based test suites. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 107–111, 2002.
- [159] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino. Know your neighbor: Fast static prediction of test flakiness. *IEEE Access*, 9:76119–76134, 2021.
- [160] S. Vysali, S. McIntosh, and B. Adams. Quantifying, characterizing, and mitigating flakily covered program elements. *Transactions on Software Engineering*, 2020.
- [161] M. Waterloo, S. Person, and S. Elbaum. Test analysis: Searching for faults in tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 149–154, 2015.
- [162] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam. Preempting flaky tests via non-idempotent-outcome tests. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.
- [163] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 270–287, 2021.
- [164] K. D. Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001.
- [165] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, 2014.
- [166] W. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *Transactions on Software Engineering*, 42(8):707–740, 2016.
- [167] H. Ye, M. Martinez, and M. Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):1–15, 2021.
- [168] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
- [169] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian. TERMINATOR: Better automated UI test case prioritization. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 883–894, 2019.

- [170] V. M Zavala and A. Flores-Tlacuahuac. Stability of multiobjective predictive control: A utopia-tracking approach. *Automatica*, 48(10):2627–2632, 2012.
- [171] C. V. G. Zelaya. Towards explaining the effects of data preprocessing on machine learning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2086–2090, 2019.
- [172] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Transactions on Software Engineering*, 28(2):183–200, 2002.
- [173] X. Zeng and T. R. Martinez. Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(1):1–12, 2000.
- [174] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page 50–61, 2021.
- [175] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396, 2014.
- [176] Y. Zhang, R. Jin, and Z. Zhou. Understanding bag-of-words model: A statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.
- [177] H. Zhu, L. Wei, M. Wen, Y. Liu, S. C. Cheung, Q. Sheng, and C. Zhou. MockSniffer: Characterizing and recommending mocking decisions for unit tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 436–447, 2020.
- [178] C. Ziftci and D. Cavalcanti. De-flake your tests automatically locating root causes of flaky tests in code at Google. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 736–745, 2020.
- [179] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience*, 51(5):851–867, 2020.
- [180] M. Fowler. Eradicating non-determinism in tests. <https://martinfowler.com/articles/nonDeterminism.html>, 2011.
- [181] P. Sudarshan. No more flaky tests on the Go team. <https://www.thoughtworks.com/en-gb/insights/blog/no-more-flaky-tests-go-team>, 2012.
- [182] android - NPE inside robotium - Stack Overflow. <https://stackoverflow.com/questions/23519395/npe-inside-robotiumk>, 2014.
- [183] J. Micco. Flaky tests at Google and how we mitigate them. <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>, 2016.
- [184] jenkins - How do you label flaky tests using junit? - Stack Overflow. <https://stackoverflow.com/questions/39538400/how-do-you-label-flaky-tests-using-junit>, 2016.
- [185] L. Eloussi. Flaky tests (and how to avoid them). <https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60>, 2016.
- [186] ruby on rails - Flaky tests with DatabaseCleaner and transactions. How to debug? - Stack Overflow. <https://stackoverflow.com/questions/37560303/flaky-tests-with-databasecleaner-and-transactions-how-to-debug>, 2016.
- [187] J. Micco. The state of continuous integration testing at Google. <http://aster.or.jp/coference/icst2017/program/jmicco-keynote.pdf>, 2017.

- [188] ruby - Can I detect if an element (button) is “clickable” in my specs? - Stack Overflow. <https://stackoverflow.com/questions/48027118/can-i-detect-if-an-element-button-is-clickable-in-my-rspecs>, 2017.
- [189] E. Dillon, J. LaRiviere, S. Lundberg, J. Roth, V. Syrgkanis. Be careful when interpreting predictive models in search of causal insights. https://shap.readthedocs.io/en/latest/example_notebooks/overviews/Be%20careful%20when%20interpreting%20predictive%20models%20in%20search%20of%20causal%20insights.html, 2018.
- [190] B. Lee. We have a flaky test problem. <https://medium.com/scopedev/how-can-we-peacefully-co-exist-with-flaky-tests-3c8f94fba166>, 2019.
- [191] K. Beck. Facebook engineering process with Kent Beck. <https://softwareengineeringdaily.com/2019/08/28/facebook-engineering-process-with-kent-beck/>, 2019.
- [192] react native - Detox: detect that element was displayed - Stack Overflow. <https://stackoverflow.com/questions/59412749/detox-detect-that-element-was-displayed>, 2019.
- [193] Stack Overflow developer survey 2021. <https://insights.stackoverflow.com/survey/2021>, 2021.
- [194] Surveying the developer experience of flaky tests: Replication package. <https://github.com/flake-it/flaky-test-survey-replication-package>, 2021.
- [195] 12. Virtual environments and packages — Python 3.11.2 documentation. <https://docs.python.org/3/tutorial/venv.html>, 2023.
- [196] 6.3. Preprocessing data — scikit-learn 1.2.2 documentation. <https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler>, 2023.
- [197] 9. Classes — Python 3.11.2 documentation. <https://docs.python.org/3/tutorial/classes.html>, 2023.
- [198] ACM digital library. <https://dl.acm.org/>, 2023.
- [199] Agitar Technologies: Putting Java to the test. <http://www.agitar.com/>, 2023.
- [200] airflow/test_local_client.py at c743b95a02ba1ec04013635a56ad042ce98823d2. https://github.com/apache/airflow/blob/c743b95a02ba1ec04013635a56ad042ce98823d2/tests/api/client/test_local_client.py#L127, 2023.
- [201] apache/airflow at c743b95a02ba1ec04013635a56ad042ce98823d2. <https://github.com/apache/airflow/tree/c743b95a02ba1ec04013635a56ad042ce98823d2>, 2023.
- [202] Assert (JUnit API). <https://junit.org/junit4/javadoc/latest/org/junit/Assert.html>, 2023.
- [203] AssertJ - fluent assertion Java library. <https://assertj.github.io/doc/>, 2023.
- [204] August Shi. <https://sites.utexas.edu/august/>, 2023.
- [205] Automatically training project-specific machine learning models to detect and classify flaky tests: Replication package. <https://github.com/flake-it/flake-friend-evaluator>, 2023.
- [206] box/flaky: Plugin for nose or pytest that automatically reruns flaky tests. <https://github.com/box/flaky>, 2023.
- [207] BuildPulse · Detect, tracks and eliminate flaky tests. <https://buildpulse.io/>, 2023.

- [208] Calendar (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Calendar.html>, 2023.
- [209] Class (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html>, 2023.
- [210] Coroutines and tasks — Python 3.11.2 documentation. <https://docs.python.org/3/library/asyncio-task.html>, 2023.
- [211] Coverage.py — Coverage.py 7.2.1 documentation. <https://coverage.readthedocs.io/en/stable/>, 2023.
- [212] Darko Marinov. <https://mir.cs.illinois.edu/marinov/>, 2023.
- [213] DateFormatSymbols (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/text/DateFormatSymbols.html>, 2023.
- [214] Docker: Accelerated, containerized application development. <https://www.docker.com/>, 2023.
- [215] Docker docs: How to build, share, and run applications. <https://docs.docker.com/>, 2023.
- [216] EC2 on-demand instance pricing – Amazon web services. <https://aws.amazon.com/ec2/pricing/on-demand/>, 2023.
- [217] Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models: Replication package. <https://github.com/flake-it/cannier-experiment>, 2023.
- [218] Evaluating features for machine learning detection of order- and non-order-dependent flaky tests: Replication package. <https://github.com/flake-it/flake16-framework>, 2023.
- [219] EvoSuite — Automatic test suite generation for Java. <https://www.evosuite.org/>, 2023.
- [220] facebookresearch/hydra: Hydra is a framework for elegantly configuring complex applications. <https://github.com/facebookresearch/hydra>, 2023.
- [221] Firefox - Protect your life online with privacy-first products — Mozilla. <https://www.mozilla.org/firefox>, 2023.
- [222] Fix flaky media player test (#36358) · home-assistant/core@2b42d77. <https://github.com/home-assistant/core/commit/2b42d77f5899b9a87b7713c4582b71bce1fd40dd>, 2023.
- [223] flake-it/cannier-framework. <https://github.com/flake-it/cannier-framework>, 2023.
- [224] flake-it/flaky-tests-bibliography. <https://github.com/flake-it/flaky-tests-bibliography>, 2023.
- [225] flake-it/pytest-cannier. <https://github.com/flake-it/pytest-cannier>, 2023.
- [226] Glossary — Python 3.11.2 documentation,. <https://docs.python.org/3/glossary.html#term-global-interpreter-lock>, 2023.
- [227] Google Chrome - Download the fast, secure browser from Google. <https://www.google.com/chrome>, 2023.
- [228] Google scholar. <https://scholar.google.com/>, 2023.
- [229] HashMap (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>, 2023.

- [230] HashSet (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>, 2023.
- [231] Home – Travis-CI. <https://travis-ci.org/>, 2023.
- [232] Home - Springer. <https://link.springer.com/>, 2023.
- [233] Home — ESEC / FSE. <https://www.esec-fse.org/>, 2023.
- [234] home-assistant/core: Open source home automation that puts local control and privacy first. <https://github.com/home-assistant/core>, 2023.
- [235] HtmlUnit - Welcome to HtmlUnit. <https://htmlunit.sourceforge.io/>, 2023.
- [236] hydra/test_compose.py at 804dab9e710496112260c42e3be57c5467b0c2dd · facebookresearch/hydra. https://github.com/facebookresearch/hydra/blob/804dab9e710496112260c42e3be57c5467b0c2dd/tests/test_compose.py#L54, 2023.
- [237] ICSE home page. <http://www.icse-conferences.org/>, 2023.
- [238] ICST steering committee. <https://cs.gmu.edu/icst/>, 2023.
- [239] IEEE Xplore. <https://ieeexplore.ieee.org/>, 2023.
- [240] imbalanced-learn documentation — Version 0.10.1. <https://imbalanced-learn.org/stable/index.html>, 2023.
- [241] I/O statistics fields. <https://www.kernel.org/doc/Documentation/iostats.txt>, 2023.
- [242] ipython/test_async_helpers.py at 95d2b79a2bd889da7a29e7c3cf5f49c1d25ff43d. https://github.com/ipython/ipython/blob/95d2b79a2bd889da7a29e7c3cf5f49c1d25ff43d/IPython/core/tests/test_async_helpers.py#L135, 2023.
- [243] J. Listfield. Google testing blog: Where do our flaky tests come from?. <https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html>, 2023.
- [244] Java Virtual Machine Tooling Interface (JVM TI). <https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/>, 2023.
- [245] Jon Bell. <https://www.jonbell.net/>, 2023.
- [246] JSON. <https://www.json.org/json-en.html>, 2023.
- [247] JSONAssert - Write JSON unit tests with less code - Cookbook. <http://jsonassert.skyscreamer.org/cookbook.html>, 2023.
- [248] JUnit 5. <https://junit.org/junit5/>, 2023.
- [249] Keyboard actions — Selenium. <https://www.selenium.dev/documentation/en/webdriver/keyboard/#sendkeys>, 2023.
- [250] Launchpad. <https://launchpad.net/>, 2023.
- [251] LinkedHashMap (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html>, 2023.
- [252] LinkedHashSet (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>, 2023.
- [253] Maven – Welcome to Apache Maven. <https://maven.apache.org/>, 2023.

- [254] Maven Surefire plugin – Rerun failing tests. <https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html>, 2023.
- [255] Michael Hilton. <https://www.cs.cmu.edu/~mhilton/>, 2023.
- [256] Mining software repositories. <http://www.msrfconf.org/>, 2023.
- [257] New EC2 M5zn instances — Fastest Intel Xeon scalable CPU in the cloud — AWS news blog. <https://aws.amazon.com/blogs/aws/new-ec2-m5zn-instances-fastest-intel-xeon-scalable-cpu-in-the-cloud/>, 2023.
- [258] Object (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>, 2023.
- [259] OpenMined/PySyft: Data science on data without acquiring a copy. <https://github.com/OpenMined/PySyft>, 2023.
- [260] ossf/criticality_score: Gives criticality score for an open source project. https://github.com/ossf/criticality_score, 2023.
- [261] PhantomJS - Scriptable headless browser. <https://phantomjs.org/>, 2023.
- [262] PIT mutation testing. <https://pitest.org/>, 2023.
- [263] Process (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>, 2023.
- [264] psutil documentation — psutil 5.7.3 documentation. <https://psutil.readthedocs.io/en/stable/>, 2023.
- [265] PyPI · the Python Package Index. <https://pypi.org/>, 2023.
- [266] pytest: helps you write better programs — pytest documentation. <https://docs.pytest.org/>, 2023.
- [267] Random (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Random.html>, 2023.
- [268] Randoop: Automatic unit test generation for Java. <https://randoop.github.io/randoop/>, 2023.
- [269] Remove test_spinup_time (#3603) · OpenMined/PySyft@fc729ac. <https://github.com/OpenMined/PySyft/commit/fc729acc88c2c26ece4fb5e4a931a7a13fe0971a>, 2023.
- [270] Saving repositories with stars - GitHub docs. <https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars>, 2023.
- [271] scikit-learn: machine learning in Python — scikit-learn 1.2.2 documentation. <https://scikit-learn.org/stable/>, 2023.
- [272] Scopus - Document search. <https://www.scopus.com/>, 2023.
- [273] Scratch - Imagine, program, share. <https://scratch.mit.edu/>, 2023.
- [274] SecurityManager (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html>, 2023.
- [275] Selenium. <https://www.selenium.dev/>, 2023.
- [276] Sheffield Digital – The association for the people and businesses of Sheffield’s digital industries. <https://sheffield.digital/>, 2023.

- [277] Thread (Java platform SE 8). <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>, 2023.
- [278] unittest — Unit testing framework — Python 3.11.2 documentation. <https://docs.python.org/3/library/unittest.html>, 2023.
- [279] Welcome to Radon’s documentation! — Radon 4.1.0 documentation. <https://radon.readthedocs.io/en/stable/index.html>, 2023.
- [280] Welcome to the Apache Software Foundation!. <https://www.apache.org/>, 2023.
- [281] Welcome to the SHAP documentation! — SHAP latest documentation. <https://shap.readthedocs.io/en/stable/index.html>, 2023.
- [282] Wing Lam’s home page. <https://mir.cs.illinois.edu/winglam/>, 2023.

Appendix A

Appendix A: Ethics Application



Application 041141

Section A: Applicant details

Date application started:
Thu 10 June 2021 at 09:50

First name:
Owain

Last name:
Parry

Email:
oparry1@sheffield.ac.uk

Programme name:
Standard PhD in Department of Computer Science

Module name:
None

Last updated:
23/06/2021

Department:
Computer Science

Applying as:
Postgraduate research

Research project title:
Developer Views on Flaky Tests: Impacts, Causes, and Actions

Has your research project undergone academic review, in accordance with the appropriate process?
Yes

Similar applications:
- *not entered* -

Section B: Basic information

Supervisor

Name	Email
Phil McMinn	p.mcminn@sheffield.ac.uk

Proposed project duration

Start date (of data collection):
Thu 1 July 2021

Anticipated end date (of project)
Sat 28 August 2021

3: Project code (where applicable)

Project externally funded?
Yes

Project code
R/155878-11-1

Suitability

Takes place outside UK?

Yes

Involves NHS?

No

Health and/or social care human-interventional study?

No

ESRC funded?

No

Likely to lead to publication in a peer-reviewed journal?

Yes

Led by another UK institution?

No

Involves human tissue?

No

Clinical trial or a medical device study?

No

Involves social care services provided by a local authority?

No

Is social care research requiring review via the University Research Ethics Procedure

No

Involves adults who lack the capacity to consent?

No

Involves research on groups that are on the Home Office list of 'Proscribed terrorist groups or organisations'?

No

Indicators of risk

Involves potentially vulnerable participants?

No

Involves potentially highly sensitive topics?

No

Section C: Summary of research

1. Aims & Objectives

The aim of our study is to understand the views of software developers regarding flaky tests. A flaky test is a software test (a piece of code that attempts to verify a property or behavior of a software system, giving a "pass" or "fail" outcome) that can pass or fail without any changes to the code being tested and is thus an unreliable indicator of correctness. Specifically, we aim to understand the impacts that flaky tests have upon software testing and development, their causes, and what actions developers take against them.

Over the past decade there have been many empirical studies of flaky tests, but relatively few have considered the views of actual software developers. Since they are the ones who actually have to deal with flaky tests, it is our position that developers are a highly valuable source of information.

2. Methodology

To answer our research questions, we plan to distribute a short Google Forms questionnaire to software developers. This will be done via Twitter, where a link to the questionnaire will be provided in a Tweet which will subsequently be retweeted by the researchers. We may decide to use other social networking services too, such as LinkedIn or Reddit (e.g., /r/programming), where the survey will be distributed

in a similar fashion (i.e., linked to in a post). Our survey is split into three sections, addressing the impacts of flaky tests, the causes of flaky tests, and what actions developers take against flaky tests. Our analysis will be mainly quantitative, with the majority of the questions being answered on a Likert scale.

3. Personal Safety

Have you completed your departmental risk assessment procedures, if appropriate?

Not applicable

Raises personal safety issues?

No

Our research will be primary conducted via an online questionnaire and thus there will be no possibility of any physical danger to the participants. With regards to mental well-being, our questionnaire is aimed at professional software developers regarding a particular problem in software testing (flaky tests) and there are no personal questions or questions of a sensitive nature involved.

Section D: About the participants

1. Potential Participants

The potential participants are software developers who have experienced flaky tests. We generally expect participants to self-identify as such by responding to our questionnaire, though we do ask how often respondents experience flaky tests and how many years experience they have in software development to verify their eligibility as participants.

2. Recruiting Potential Participants

Since there is a highly active body of software developers using Twitter, we plan to distribute our questionnaire mainly via a link within a Tweet. As explained in the previous section, we may also disseminate the survey via a link within posts on other social networking services such as Reddit. Participants will "recruit" themselves to the study by answering the questionnaire, which they will be made aware of before answering any questions. To motivate potential participants to join the study, we will summarize the potential benefits of the research to the software engineering community in the Tweets/posts.

2.1. Advertising methods

Will the study be advertised using the volunteer lists for staff or students maintained by IT Services? No

- not entered -

3. Consent

Will informed consent be obtained from the participants? (i.e. the proposed process) Yes

The first page of the questionnaire informs potential participants about how their responses will be used in the research and explicitly asks for their consent. Participants will be unable to continue with the questionnaire without giving consent.

4. Payment

Will financial/in kind payments be offered to participants? No

5. Potential Harm to Participants

What is the potential for physical and/or psychological harm/distress to the participants?

None.

How will this be managed to ensure appropriate protection and well-being of the participants?

Not applicable.

6. Potential harm to others who may be affected by the research activities

Which other people, if any, may be affected by the research activities, beyond the participants and the research team?

None.

What is the potential for harm to these people?

Not applicable.

How will this be managed to ensure appropriate safeguarding of these people?

Not applicable.

7. Reporting of safeguarding concerns or incidents

What arrangements will be in place for participants, and any other people external to the University who are involved in, or affected by, the research, to enable reporting of incidents or concerns?

The first point of contact for those wanting to report an incident or concern is the Principal Investigator (Owain Parry) whose details we make available in the Participant Information Sheet. We also give the contact details of the Head of the Department of Computer Science (Guy Brown), should the person raising the concern, participant or otherwise, feel that their concern has not been handled appropriately.

Who will be the Designated Safeguarding Contact(s)?

Owain Parry (oparry1@sheffield.ac.uk)

How will reported incidents or concerns be handled and escalated?

In the first instance, the Principal Investigator will attempt to resolve concerns. For example, if a participant wishes to withdraw from the study then the Principal Investigator can remove their responses from the data set. If the Principal Investigator is unable to resolve a concern alone, then they will involve the other members of the research team who will discuss and attempt to resolve the concern collectively. If the research team is unable to resolve a concern, then they will escalate it within the Department of Computer Science, though to whom specifically would depend on the nature of the concern.

Section E: About the data

1. Data Processing

Will you be processing (i.e. collecting, recording, storing, or otherwise using) personal data as part of this project? (Personal data is any information relating to an identified or identifiable living person).

Yes

Which organisation(s) will act as Data Controller?

University of Sheffield only

2. Legal basis for processing of personal data

The University considers that for the vast majority of research, 'a task in the public interest' (6(1)(e)) will be the most appropriate legal basis. If, following discussion with the UREC, you wish to use an alternative legal basis, please provide details of the legal basis, and the reasons for applying it, below:

- *not entered* -

Will you be processing (i.e. collecting, recording, storing, or otherwise using) 'Special Category' personal data?

No

3. Data Confidentiality

What measures will be put in place to ensure confidentiality of personal data, where appropriate?

Personal data will be stored separately from the main data set and will only be accessible to the members of the research team from the University of Sheffield. We will also screen the main data set to ensure no participants have included personal data in their answers, in which case they will be withdrawn from the study and their responses discarded. Only the main data set after screening will be analyzed and be made publicly available, ensuring all participants are anonymous.

4. Data Storage and Security

In general terms, who will have access to the data generated at each stage of the research, and in what form

The only researchers who will have access to any personal data will be those from the University of Sheffield. This will be solely in the form of participants' email addresses, which they have the option of providing.

What steps will be taken to ensure the security of data processed during the project, including any identifiable personal data, other than those already described earlier in this form?

Any personal data will be submitted by participants via Google Forms where it is automatically compiled into a Google Sheets spreadsheet. As part of the University of Sheffield's account security policy, the researchers who will have access to this data have two-factor authentication enabled on their Google accounts.

Will all identifiable personal data be destroyed once the project has ended?

Yes

Please outline when this will take place (this should take into account regulatory and funder requirements).

The only personal information that will be collected from participants will be their email addresses, which they have the option of providing. This will be solely used to inform the participants of the results of the research and will be stored separately from the main data set. These records will be destroyed immediately after publication.

Section F: Supporting documentation

Information & Consent

Participant information sheets relevant to project?

Yes

[Document 1093199 \(Version 1\)](#)

[All versions](#)

Consent forms relevant to project?

Yes

[Document 1093109 \(Version 2\)](#)

[All versions](#)

Additional Documentation

External Documentation

Questionnaire: <https://docs.google.com/forms/d/e/1FAIpQLSd9XBBEF6fdC-AffWeeEMX-sQDHYS4WHRICtbrjVYxQa35ilQ/viewform>

Section G: Declaration

Signed by:

Owain Parry

Date signed:

Tue 22 June 2021 at 14:42

Official notes

- not entered -