UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2023-064
The Faculty of Sciences, Technology and Medecine

Dissertation

Presented on June 29th, 2023 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Guillaume Haben
Born on 14th November 1993 in Essey-lès-Nancy (France)

Test Flakiness Prediction Techniques for Evolving Software Systems

Dissertation defence committee

God does not play dice with the universe.
—ALBERT EINSTEIN, The Born-Einstein Letters
1916-55

Abstract

Software testing plays a crucial role to guarantee a desired level of software quality. Its goal is to ensure software products respect specified requirements, function as intended and are error-free. The scope of software testing is broad, from functional to non-functional requirements, and is generally performed at different levels (*e.g.* unit testing, integration testing, system testing, acceptance testing). During continuous integration, development activities are typically stopped when test failures happen and further investigations and debugging are required.

In an ideal world, all tests are deterministic: developers and testers expect the same outcome (pass or fail) for a test when executed twice on the same version of their program. Unfortunately, some tests exhibit non-deterministic behaviour. Commonly named flaky tests, they give confusing signals to developers who struggle to understand if their software is defective or not, and tend to lower their trust in test suites. Those occasional test failures can be difficult to reproduce and thus hard to debug. When test flakiness is left unaddressed, it can hinder the smooth and rapid integration of code changes. Furthermore, it also impacts many effective testing techniques such as test case selection, test case prioritisation, automated program repair or fault localisation. While the phenomenon is known by many for decades now, academic attention has only sprouted in recent years and few studies were carried out to better understand flakiness, its different causes and origins, and to propose techniques helping to prevent, detect and mitigate flakiness.

In this context, the present dissertation aims at advancing research in test flakiness prediction through five main contributions. The first two are explorative studies. They aim at getting a better understanding of test flakiness and existing prediction techniques. The next two contributions are constructive studies. They suggest new approaches and focus on yet unaddressed problems. Finally, the last contribution is a case study carried out in a real-world context and brings new insights important to efficiently continue test flakiness prediction research.

By conducting a qualitative study, the first contribution seeks to understand practitioners' perceptions of the sources, impact and mitigation strategies of flaky tests. The goal of this work is to grasp the current challenges revolving around flakiness in the industry and to identify opportunities for future research. We carried out this study by conducting a grey literature review and practitioner interviews. Findings revealed sources of flakiness that were until now overlooked by previous research (such as the infrastructure, environment or testing frameworks) and a strong negative impact on testing practices. The second contribution aims at comforting the usability of flaky test prediction techniques. Rerunning failing tests is still the main approach to deal with flakiness and this comes at a cost, both time-wise and computer-wise. If accurate, predicting flaky tests can be an alternative to reruns and help better understand their characteristics. In this study, we replicate an existing approach relying on code vocabulary to predict flaky tests with three goals in mind: validating the approach in the continuous integration context, evaluating the generalisability of the approach to different programming languages and extending the approach by considering an additional set of features.

Realising that predicting flaky tests is feasible but also that challenges remain to understand the cause of flakiness, the third contribution presents a new technique to predict the flakiness category of a given flaky test with the hope to provide developers with better insights to debug their tests. In the fourth contribution, we aim at identifying the cause of flakiness in the critical case where its source originates from within the program under test. To do so, we adapt spectrum-based fault localisation and leverage ensemble learning to rank classes based on their likelihood to be responsible for test flakiness.

In the fifth and final contribution, we conduct an empirical analysis of Chromium's continuous integration, where we found that flaky test signals should not be discarded as they reveal themselves useful to find faults caused by regressions. Thus, we advocate for the need to predict failures (as flaky or faulty) by taking into account the context of a test's execution.

Overall, this thesis provides insights into how predictive models can be validated and leveraged to better handle test flakiness in real-world contexts.

Acknowledgments

Embarking on the arduous yet exhilarating journey of a PhD, one finds himself traversing an emotional rollercoaster of intellectual growth, self-discovery, and relentless perseverance. Approaching its conclusion, I realise now how well-surrounded I was throughout the four years I spent in the SerVal team of the SnT research centre. Here, I want to say a few words to those who took an important part in this adventure and had positive impacts on it.

Yves, these first words go for you. As my official supervisor, you trusted me since the very first time we met and you gave me all the keys I needed to successfully pursue my PhD. I am thankful for your support and your constant positive thinking.

Mike, you were my direct supervisor and when I joined the team I didn't fully measure its excellence. You are largely contributing to it and I learned so much from you. I am grateful for all your counselling, help and every one of the discussions we had. I admire the amount of work you give to create this wonderful research environment.

Sarra, our paths met at a crucial point in my life. I was not expecting your arrival in our team nor that I would benefit from the additional support and expertise of a young researcher. I am deeply thankful for your guidance and cooperation. You genuinely inspired me and will continue to do so for a long time.

Maxime, while you were expanding the domains of expertise in our team, you always remained available and supportive to me. I sincerely appreciate your dedication and I will keep good memories of the discussions we had.

I would like to thank all my co-authors for their involvement and feedback in the different projects. Thanks also to the jury members for the time and interest they gave to my research. This dissertation would not have been the same without your help. I also want to express my gratitude towards all my fellow SerVal colleagues, some arrived, and some left through my journey but all contributed to creating a fantastic place to conduct research. I hope we will stay in touch.

On a more personal note, I wish to thank my family for their support: my mom, who listened so many times to my work-related stories, my dad, who doesn't know how important he is to me and my brother, whose presence has the magical ability to always transport me back to the carefree days of childhood. You are the joy of my life. Finally, I thank all my friends with whom I was able to temper the most stressful part of this adventure.

Contents

1 Introduction

This chapter provides an introduction to software testing. In particular, it explains how today's software systems are engineered, why software quality is an important matter and how software testing is commonly conducted. It also introduces the challenges encountered by developers, testers and researchers particularly those linked with flakiness; which corresponds to the scope of this thesis.

Contents

1.1 Context

Throughout the years, our civilisation become more and more dependent on computers. The laptop we use to access the internet is just the tip of the iceberg. Actually, software is now present in every aspect of our life: it's on our wrists, driving our cars, flying our planes, powering our electricity and running our economy. Software is flexible, it can be leveraged to answer very specific needs and is deployed in various forms, *e.g.* embedded in every object of the Internet of Things, where power consumption is often a key challenge, or distributed across multiple platforms such as blockchain, where traceability and reliability are expected. This shift in our society happened fast. Over the last decades, the software industry boomed, relying on more powerful hardware, and answering the need of more people.

1.1.1 Software Development Life Cycle

Traditional Software Development

The software development life cycle of software systems has evolved through the years to adapt to technical practices and customer needs. In the early days, the Waterfall model was widely used. It followed a linear and sequential process of the different phases of a project, each stage being completed before the beginning of the next one. The typical phases consisted in identifying the requirements, designing the software, implementing it, then testing and verifying and finally deploying and maintaining. Even though many consider the Waterfall model as a practice from the past, it is still adequate and efficient to rely on thanks to its clear structure in the case of small projects or when deliverables are easily defined at the beginning. Limits for this model arise for bigger projects, or when customer needs are complex and evolving. While benefiting from a stronger and longer operational lifetime, projects following the Waterfall model suffer from the difficulty of making changes as they would require whole new iterations of the different phases. Over the years, the user was gradually put at the centre of attention by software companies and constant feedback required a more flexible approach to software development [1]–[3].

Agile Software Development

Agile software development emerged as an answer to the limitations of traditional, sequential approaches like the Waterfall model. The need for a more flexible and adaptive approach arose due to the increasing complexity of software projects, the desire for faster delivery, and the recognition that user requirements often evolve during development. One of the main benefits of Agile methodology is its ability to deliver quickly and frequently. This is made possible by breaking down the whole development into several software components that can be deployed easily. This enables fast and continuous customer feedback, ensuring that the software meets evolving needs and expectations. Agile also promotes a collaborative team culture through the means of tools used for better communication and knowledge sharing across different teams. This leads to higher team motivation and morale which aims at a better quality output.

Nowadays, Agile is vastly used and different frameworks and practices exist to implement Agile principles, such as Scrum, Kanban and Extreme programming. The goals of Agile remain the same: to foster collaboration, adaptability, and customer satisfaction while enabling development teams to deliver high-quality software efficiently [4]–[7].

Continuous Integration & Continuous Delivery

Continuous Integration (CI) and Continuous Delivery (CD) are practices used in software development to automate the process of building, testing, and delivering software. They promote efficiency, quality, and agility in the development lifecycle.

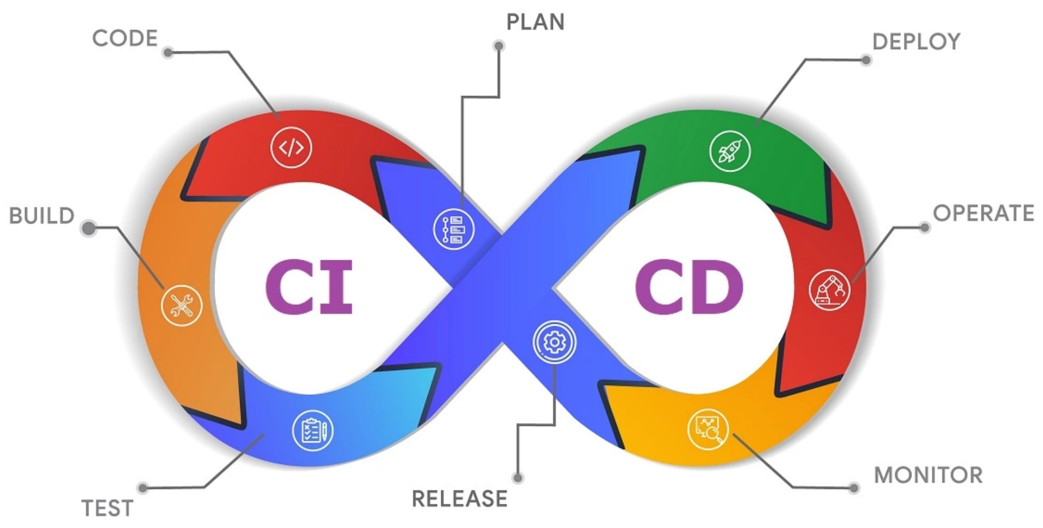


Figure 1.1: Continuous Integration and Continuous Delivery [8]

CI refers to the practice of frequently integrating code changes from multiple developers into a shared repository. The integrated code pieces are then automatically built and tested to ensure no conflict or issue emerges. This process ensures that the codebase remains stable and allows development teams to identify and fix issues early on. CI promotes collaboration, reduces the risk of integration problems, and enables rapid feedback, enabling the delivery of higher-quality software.

CD extends the concept of CI by automating the software delivery aspects. Continuous Delivery enables safe, quick and sustainable shipment of changes into production. Those changes are typically new features, bug fixes, and configuration changes. CD relies on code that is always in a deployable state in order to make deployments as easy and straightforward to facilitate automation. Figure 1.1 summarises the different steps of CI/CD.

CI/CD is closely linked to Agile methodologies. Both CI and CD foster the principles of frequent collaboration, iterative development, and continuous improvement. By automating repetitive and error-prone tasks, CI/CD enables teams to focus on delivering value, responding to changing requirements, and maintaining a sustainable pace. It aligns with Agile's emphasis on delivering working software quickly and adapting to feedback and evolving customer needs [9]–[12].

1.1.2 Software Quality Assurance

Software development methodologies evolved throughout the years. In this context, one key aspect is also evolving accordingly: software quality. Users of any application have stronger expectations over time. They require not only bug-free products but also cheap, fast, secure, easy-to-use and respectful towards their privacy. For any business now, software quality is of utmost importance. It saves time and money, as repairing bugs or patching security issues too late can be very costly, it ensures competitiveness in the market and is also key to maintaining a good brand reputation among customers. This is when the role of Software Quality Assurance comes into place [13]–[15].

Software Quality Assurance (SQA) represents any set of methods, processes and activities applied during the life cycle of a software project to guarantee its quality, reduce and prevent defects during its development and confirm its alignment with the defined requirements. SQA is defined by many standards and norms across the industry, but several principles prevail:

Prevention: The cost of software bugs is known to grow exponentially the longer it takes to be fixed. It is then important to allocate the necessary efforts to identify potential issues early in the development life cycle and to reduce the risk of shipping faulty products.

Continuous improvement: SQA is not a one-time check. It is a continuous process that needs to be followed at all times. It is also often required to adapt methodologies and processes in parallel with the evolution of the software especially at scale.

Stakeholder involvement: To achieve good quality assurance, every stakeholder needs to be involved in the process, this includes managers, developers, testers, but also the customers or users. Good communication between all parties facilitates quick feedback and enables fast reactions.

Risk prioritisation: SQA involves identifying and managing risks that could impact the quality of the software and taking proactive steps to mitigate them.

According to these principles, SQA activities are then defined and followed in a continuous manner.

Planning: The first activity should be to clearly define quality standards that the end product should meet. This includes specifications, acceptance criteria and performance metrics. These blueprints should also indicate who is responsible for each task.

Reviews: Through the iterations of the software development life cycle, reviews should be carried out to prevent software defects. This includes code reviews, writing documentation, and checking the requirements. If possible, this should be done by internal and external teams.

Testing: SQA involves testing and validating the software. Performing multi-level testing is important to ensure better quality. This includes unit tests, integration tests, system tests and acceptance tests.

Monitoring: Measuring and monitoring coding activities is important to keep track of quality indicators. This can be done using code coverage, bug-tracking systems and testing tools.

This dissertation focuses towards the software testing aspect.

1.1.3 Software Testing

Principles

Software testing is a crucial and necessary aspect of software development ensuring the quality, reliability, and functionality of software systems. It involves systematically checking and evaluating the software against predefined requirements to align with user needs and expectations. The primary goal of software testing is to find bugs, defects or errors as early as possible in the development life cycle. According to some studies [16], the cost of fixing a bug is growing exponentially the later it is found in the different phases of development and can go up to 100 times more costly if identified in production.

There are several key principles that guide software testing:

Exhaustiveness: Testing aims to achieve a high level of coverage, ensuring that all relevant functionalities, scenarios, and input combinations are tested. While it may not be possible to test every single possibility, testing efforts should be comprehensive to minimize the risk of undiscovered defects.

Independence: Testing should be conducted independently of the development process to ensure impartiality. Testers should have a clear understanding of the software's functionality but should maintain a separate perspective to identify potential issues or deviations from requirements.

Early Start: Testing should commence as early as possible in the software development life cycle. By incorporating testing from the initial stages, defects can be identified and resolved quickly, reducing the potential impact on future development phases.

Reproducibility: Tests should be reproducible, allowing defects to be isolated and fixed reliably. This principle ensures that identified issues can be accurately communicated and resolved by the development team.

Traceability: Testing should be traceable, meaning that test cases are linked to requirements and documented in a structured manner. This enables effective tracking of the testing process and ensures coverage of specific requirements.

Continuous Improvement: Testing should be viewed as an iterative and continuous process. It involves learning from past experiences, analyzing test results, and refining testing strategies to enhance efficiency and effectiveness over time. Continuous improvement is key to optimizing the testing process and delivering higher-quality software.

By adhering to these principles, software testing helps to identify defects, enhance software quality, and ensure that software systems meet user requirements and expectations. It plays a vital role in reducing risks, enhancing customer satisfaction, and enabling the successful deployment of reliable software products.

Levels of testing

Testing is usually performed at different levels of granularity. The main type of tests are listed below:

Unit testing focuses on verifying the functionality of individual components or units of code. It is typically conducted by developers and aims to ensure that each unit behaves as intended. Mock objects or stubs may be used to isolate units for testing.

Integration testing verifies the interactions between different modules or components when they are combined. It aims to identify defects that arise from the integration of various units and ensures the proper functioning of the software as a whole.

System testing involves testing the complete and integrated system against the specified requirements. It verifies the system's compliance with functional and non-functional requirements, such as performance, security, and usability. System testing is typically performed by a dedicated testing team.

End-to-end (E2E) testing validates the entire software system as a whole, ensuring that all components work together as expected. It simulates real-life user scenarios to identify any issues or defects that may arise from the interaction and integration between different parts of the system.

Figure 1.2 represents the pyramid of tests illustrating the various type of testing. The concept of the pyramid is used to describe metaphorically the need for smaller tests and fewer high-level tests to facilitate the development and QA team to create and maintain high-quality software. Smaller tests like unit and integration tests are typically fast and easy to automate and maintain. Higher-level tests like System tests and E2E tests tend to be larger, more brittle and require more resources to write and to run. This is why it is generally advised to follow this pyramid-shaped structure when writing tests and to avoid the cupcake or ice-cream cone structure in order to facilitate the maintainability and efficiency of software testing. This list is non-exhaustive and, depending on the projects, other types of tests could be represented, such as Graphical User Interface (GUI) tests, to test visual components in the case of graphical applications, API tests, testing the different API endpoints or even manual testing, which is usually conducted occasionally or systematically on the overall system [17].

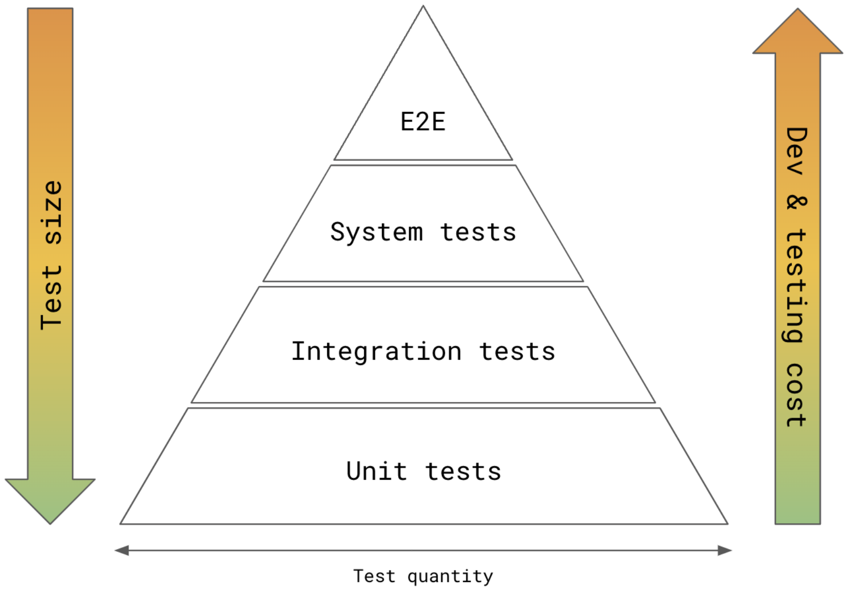


Figure 1.2: Pyramid of tests

Test coverage

One of the techniques used to assess the completeness of the testing process is test coverage. It is typically expressed as a percentage indicating the proportion of code or requirements covered by the tests. It can be measured at different test levels and can rely on different approaches, such as statement coverage, branch coverage or function coverage. Test coverage helps identify untested or under-tested areas of the software, enabling teams to focus their testing efforts and improve the overall quality of the system. However, it's important to note that achieving 100% test coverage doesn't guarantee the absence of defects, as it's still possible to have undiscovered issues in the parts that are tested [18], [19].

1.2 Challenges

1.2.1 Testing at Scale

Challenges arise when software systems become large and complex over time. It is not surprising to find projects containing several hundreds of thousands of tests in big tech companies such as Google or Meta [20], [21]. In such extreme contexts, where features are added at a very high rate (Google reports mention tens of thousands of commits per day [22]) it becomes incredibly difficult to keep testing thoroughly as running all tests for every change would require too much time and resources. Even in optimal scenarios, final results can take several hours to compute before being sent for feedback to developers.

To alleviate issues linked with important time and computing resources required, several optimisation techniques can be leveraged:

Test parallelisation relies on executing tests by dividing the test suite into smaller subsets and running them simultaneously in parallel test environments. Distributing test executions across multiple machines leveraging their combined resources can help expedite the testing process and reduce feedback time.

Commit batching involves grouping multiple code changes into a single test run, thereby reducing the number of test executions and the overall testing time. Commit batching is a strategy that can help to find a good balance between timely feedback and testing efficiency, especially when dealing with large test suites and resource constraints environments. For example, if 10 commits are batched together and result in no test failure, then we have tested 10 commits with one execution thus saving nine executions. However, if a failure occurs it is not straightforward to know which build caused the failure. A bisection is then run to isolate the failing build. Further investigations to identify the culprit build and root cause remain challenging. Research works have been carried out to reduce the search space of bug-inducing commits [23]–[25].

Test Selection is another technique that can be used to reduce the number of tests to execute. In this case, only the relevant tests are executed based on code changes or impacted areas. This reduces the number of tests run and speeds up the feedback cycle. This technique is commonly referred to as Regression Test Selection (RTS) and, even if it has been investigated for many years - several surveys date back to the 2000s [26]–[28] - it remains a challenge to optimise test selection in CI [29] and to find efficient ways to select tests.

Test Prioritisation can also be leveraged to reduce testing costs and feedback time. Commonly referred to as Test Case Prioritisation (TCP), the idea consists of starting to test the critical tests or the ones having a strong ability to identify defects. Executing high-priority tests first helps to get rapid feedback. This approach ensures that important issues are identified early in the testing process. As an example, if only one test will fail for a given test suite execution, it is then better to have this test executed among the first ones.

1.2.2 Automated Testing Research

Many research directions are taken to help test more thoroughly software and to help with debugging tasks in a more automated and efficient way. The list below presents some of the main aspects drawing attention nowadays:

Mutation Testing is a type of software testing used as a test adequacy criteria, *i.e.* to assess the effectiveness and thoroughness of existing tests. It does so by introducing artificial faults, small syntactic changes, into the original program and then checking the ability of existing test suites to detect those changes. As we saw earlier, test coverage only cannot guarantee that a program is tested thoroughly. The challenge for mutation testing is that it can be computationally expensive and time-consuming, especially for large codebases. Many works try to reduce the number of mutants [30], [31] (*e.g.* finding equivalent mutants), to identify and select relevant mutants [32]–[34] and to evaluate their applicability in the software evolution context [35], [36].

Automated Test Generation can reduce the effort and time required to create new test cases during the development process. Tools and frameworks exist to generate relevant test cases with regards to the targeted source code [37]–[39]. Different are commonly used for that purpose: code-based, property-based, random-based... One of the biggest challenge in automated test generation is linked with the test oracle problem: tools often face the challenge of determining the expected output or behavior of the system under test. Without a clear oracle, it becomes difficult to evaluate whether the generated tests are correct or not.

Fault Localisation is a research area in software testing that focuses on identifying the specific locations of software defects. It aims to assist developers in quickly identifying the parts of the code that are responsible for the observed failures. Different approaches exist [40]. Spectrum-based approaches leverage information from the program's execution trace or coverage data to pinpoint potential fault locations [41], [42]. These approaches compute suspiciousness scores for program elements (e.g., statements, branches) based on the difference between passing and failing executions. Search-based approaches formulate fault localisation as an optimisation problem and employ search algorithms to explore the space of possible fault locations [43]. Machine learning approaches use statistical analysis or machine learning models to identify faults [44]. They leverage various code and program characteristics, such as code metrics, static analysis results or historical data. Common challenges in automated fault localisation research include the accuracy of fault localisation results, the scalability to handle large codebases or the generalizability across different software contexts.

Automated Program Repair aims at automatically patching defects. Similarly to fault localisation, APR includes different approaches, like search-based repair, constraint-based repair or pattern-based repair. Challenges remain to evaluate patch correctness and quality, to have tools scaling with the size and complexity of the software system and to promote the adoption of such approaches in real-world context [45]–[47].

1.2.3 Test Flakiness

Most, if not all, of the approaches presented above, rely on deterministic tests and execution: when a given input results in a program state, we expect the output to be reproducible with similar input. This is not always the case, and developers often face what they call *flaky tests*. Flaky tests are tests that pass and fail for the same code under test.

It complicates the debugging One of the most common actions taken to debug a test that fails is to rerun it. Unfortunately in the case of a flaky test, it might be difficult to reproduce the error. Developers would then understand that something is wrong without figuring out what. This augments the time and effort given to debug and solve flakiness [48].

It reduces developers trust Flakiness is not always under the control of developers. A test might be flaky because of external dependencies not available at the time of execution or because of infrastructure issues. In this case, developers would tend to ignore signals from their tests. This can lead to real bugs being disregarded and ultimately results in less quality in the software [49].

It alters existing techniques Outside the direct challenges faced by developers, flakiness can also impact automated software techniques. As we saw earlier, many automated testing approaches can be implemented to reduce testing costs. The presence of flakiness brings noise altering the efficiency of those approaches [50], [51].

It critically affects the industry Flakiness is reported to be one of the major software testing challenges encountered nowadays. Several reports from the industry highlight this [21], [52]–[56]. For instance, at Google, there are 150 million test executions per day, and almost 16% of their 4.2 million test cases have some level of flakiness [20]. This entails enormous computational resources since 2-16% of the company's testing budget is dedicated to rerunning flaky tests [56]. Perhaps worse, over 80% of observed transitions (false Failures or Passes) at Google workflow are caused by flaky tests [57], indicating an important level of uncertainty in the test signal.

The next chapter will give more details about the origins of flakiness and some examples.

1.3 Scope of the Thesis

This dissertation largely focuses on the use of machine learning techniques to predict different aspects of flakiness, such as if a test is flaky or not, which part of the code under test is causing the flakiness, or which category of flakiness a flaky test belongs to. It also intends to better understand and assess their usability and validity in real-world scenarios. The subjects used for the different studies are coming from the open-source community. It is also worth noting that the studies mainly focus on automated functional testing: unit tests, integration tests and GUI tests but do not specifically address higher-level tests such as system tests or even manual tests as there tend to be fewer of them in open-source software.

1.4 Overview of the Contribution and Organisation of the Dissertation

This section presents the contributions of this dissertation to address the aforementioned challenges related to flakiness as well as the organisation of this dissertation as illustrated by Figure 1.3.

1.4.1 Contributions

The contributions of this dissertation are the following:

- **Chapter 4: A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests** We performed a grey literature review and interviewed 14 practitioners in order to have a better understanding of the challenges linked with flakiness in the industry. We explore three aspects: the sources of flakiness within the testing ecosystem, the impacts of flakiness and the measures adopted when addressing flakiness. Our analysis showed that, besides the tests and code, flakiness stems from interactions between the system components, the testing infrastructure, and external factors. We also highlighted the impact of flakiness on testing practices and product quality and showed that the adoption of guidelines together with a stable infrastructure are key measures in mitigating the problem. Furthermore, we also identified automation opportunities enabling future research works.

- **Chapter 5: A Replication Study on the Usage of Code Vocabulary to Predict Flaky Tests** Recent research works explored the possibility of detecting flaky tests using supervised learning. However, to reach industrial adoption and practice, these techniques need to be replicated and evaluated extensively on multiple datasets, occasions and settings. In view of this, we perform a replication study of a recently proposed method that predicts flaky tests based on their code vocabulary. We thus replicate the original study on three different dimensions. First, we replicate the approach on the same subjects as in the original study but using a different evaluation methodology, *i.e.* we adopt a time-sensitive selection of training and test sets to better reflect the envisioned use case. Second, we consolidate the findings of the original study by checking the generalisability of the results for a different programming language. Finally, we propose an extension to the original approach by experimenting with different features extracted from the code under test.

- **Chapter 6: Predicting Flaky Tests Categories using Few-Shot Learning** While promising, existing flakiness detection approaches mainly focus on classifying tests as flaky or not and, even when high performances are reported, it remains challenging to understand the cause of flakiness. This part is crucial for researchers and developers that aim to fix it. To help with the comprehension of a given flaky test, this chapter introduces FlakyCat, the first approach to classify flaky tests based on their root cause category. FlakyCat relies on CodeBERT for code representation and leverages Siamese networks to train a multi-class classifier. We train and evaluate FlakyCat on a set of 451 flaky tests collected from open-source Java projects. Our evaluation shows that FlakyCat categorises flaky tests accurately, with an F1 score of 73%. We also investigate the performance of FlakyCat for each category. In addition, to facilitate the comprehension of FlakyCat's prediction, we present a new technique for CodeBERT-based model interpretability that highlights code statements influencing the categorisation.

- **Chapter 7: Pinpointing Classes Responsible for Test Flakiness** To mitigate the effects of flakiness, both researchers and industrial experts proposed strategies and tools to detect and isolate flaky tests. However, flaky tests are rarely fixed as developers struggle to localise and understand their causes. Additionally, developers working with large codebases often need to know the sources of nondeterminism to preserve code quality, *i.e.* avoid introducing technical debt linked with non-deterministic behaviour, and avoid introducing new flaky tests. To aid with these tasks, we propose re-targeting

Fault Localisation techniques to the flaky component localisation problem, *i.e.* pinpointing program classes that cause the non-deterministic behaviour of flaky tests. In particular, we employ Spectrum-Based Fault Localisation (SBFL), a coverage-based fault localisation technique commonly adopted for its simplicity and effectiveness. We also utilise other data sources, such as change history and static code metrics, to further improve the localisation. Our results show that augmenting SBFL with change and code metrics ranks flaky classes in the top-1 and top-5 suggestions, in 26% and 47% of the cases. Overall, we successfully reduced the average number of classes inspected to locate the first flaky class to 19% of the total number of classes covered by flaky tests. Our results also show that localisation methods are effective in major flakiness categories, such as concurrency and asynchronous waits, indicating their general ability to identify flaky components.

- **Chapter 8: The Importance of Discerning Flaky from Fault-triggering Test Failures** While promising, the actual utility of the methods predicting flaky tests remains unclear since they have not been evaluated within a continuous integration (CI) process. In particular, it remains unclear what is the impact of missed faults, *i.e.* the consideration of fault-triggering test failures as flaky, at different CI cycles. In this chapter, we apply state-of-the-art flakiness prediction methods at the Chromium CI and check their performance. Perhaps surprisingly, we find that the application of such methods leads to numerous faults missed, which is approximately $^3/_4$ of all regression faults. To explain this result, we analyse the fault-triggering failures and find that flaky tests have a strong fault-revealing capability, *i.e.* they reveal more than $^1/_3$ of all regression faults, indicating inevitable mistakes of methods that focus on identifying flaky tests, instead of flaky test failures. We also find that 56.2% of fault-triggering failures, made by non-flaky tests, are misclassified as flaky. To deal with these issues, we build failure-focused prediction methods and optimize them by considering new features. Interestingly, we find that these methods perform better than the test-focused ones, with an MCC increasing from 0.20 to 0.42. Overall, our findings suggest that future research should focus on predicting flaky test failures instead of flaky tests (to reduce missed faults) and reveal the need for adopting more thorough experimental methodologies when evaluating flakiness prediction methods (to better reflect the actual practice).

1.4.2 Organisation of the Dissertation

In the remainder of this dissertation, Chapter 2 presents the technical background useful for a good understanding of this thesis. Chapter 3 discusses the existing works related to the contributions of this dissertation. Chapters 4 and 5 both contribute to the exploratory studies. The former presents a qualitative study on the sources, impacts and mitigation strategies of flakiness, and the latter details a replication study on the usage of code vocabulary to predict flaky tests. Chapters 6 and 7 are constructive studies. Chapter 6 presents FlakyCat, the first approach to classify flaky tests according to their category of flakiness and Chapter 7 introduces an approach to identify classes responsible for test flakiness. Chapter 8 is the last contribution of this dissertation. We present the case study of Chromium's continuous integration and highlight the importance of discerning flaky from fault-triggering test failures. Finally, Chapter 9 concludes the dissertation and presents future research directions.

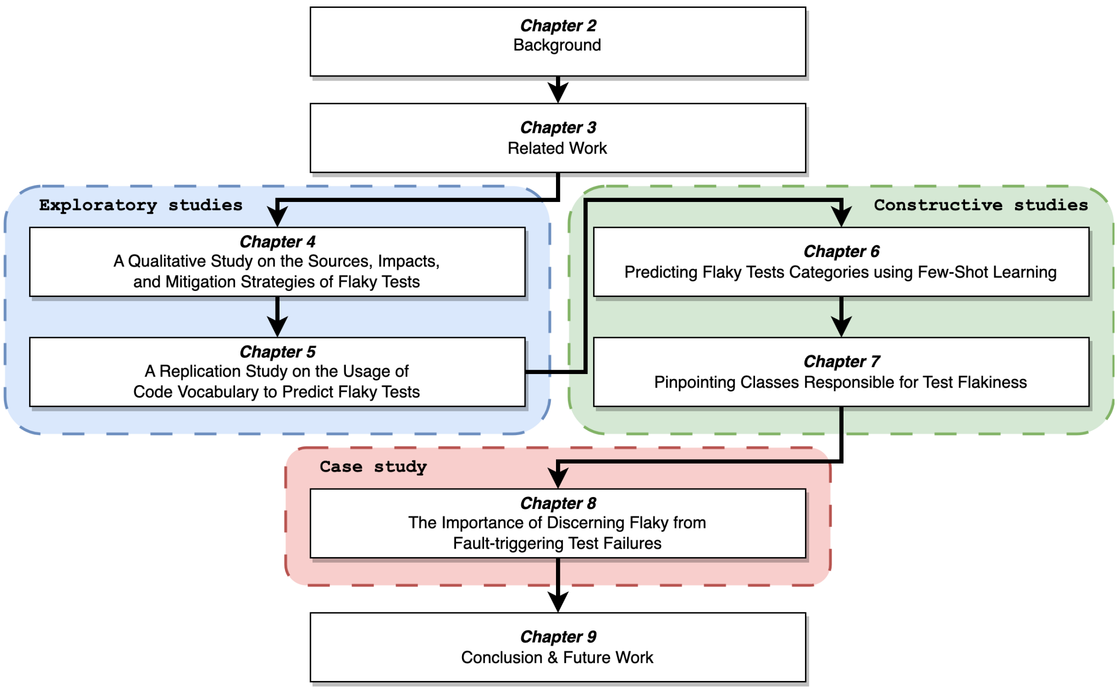

Chapter 2
Background
Chapter 3
Related Work
Exploratory studies
Chapter 4
A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests
Chapter 5
A Replication Study on the Usage of Code Vocabulary to Predict Flaky Tests
Constructive studies
Chapter 6
Predicting Flaky Tests Categories using Few-Shot Learning
Chapter 7
Pinpointing Classes Responsible for Test Flakiness
Case study
Chapter 8
The Importance of Discerning Flaky from Fault-triggering Test Failures
Chapter 9
Conclusion & Future Work

Figure 1.3: Structure of this thesis.

2 Background

This chapter presents the background on test flakiness, machine learning and spectrum-based fault localisation, essential for a good understanding of the different technical aspects of this dissertation.

Contents

2.1 Test Flakiness: Definition, Characteristics and Examples

2.1.1 Definition

In English, we refer as *flaky*, a person that is unreliable or someone that is behaving in a way that is strange, not responsible or not expected. This explains the origin of the appellation flaky test. While generally used by developers and practitioners to characterize tests that intermittently fail for no apparent reason, a general definition for a flaky test is commonly adopted by the research community:

A flaky test is a test that passes and fails when executed multiple times on the same code.

2.1.2 Examples

```java
@Test
public void testRsReportsWrongServerName() throws Exception {
  MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
  MiniHBaseClusterRegionServer firstServer =
    (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
  HServerInfo hsi = firstServer.getServerInfo();
  firstServer.setHServerInfo(...);

  // Sleep while the region server pings back
  Thread.sleep(2000);
  assertTrue(firstServer.isOnline());
  assertEquals(2,cluster.getLiveRegionServerThreads().size());
  ... // similarly for secondServer
}
```

Figure 2.1: Example of a flaky test caused by an asynchronous wait [58]

Figure 2.1 shows a unit test written in Java for the HBase project. This test is flaky and was reported by Luo *et al.* in their empirical study. We can see that the test uses a CLUSTER to initiate the server FIRSTSERVER. Then, it waits for the server to ping back by using an asynchronous wait with THREAD.SLEEP(2000). We know that this test was sometimes passing and sometimes failing depending on how fast the server was put online.

Figure 2.2 shows another flaky test taken in Elastic-Job, another popular Java project on GitHub. This test does not initially appear to be flaky, however, it was

```
1  public void assertIsShutdownAlready() {
2   shutdownListenerManager.new
         InstanceShutdownStatusJobListener().dataChanged("/
         test_job/instances/127.0.0.1@-@0", Type.
         NODE_REMOVED, "");
3   verify(schedulerFacade, times(0)).shutdownInstance();
4  }
```

Figure 2.2: Example of an order-dependent flaky test caused [59]

found by the iDFlakies tool as an order-dependent test. In their paper, Lam *et al.* explain that this test is checking that an instance of a class variable is shut down on line 3. It happens that this instance is started by another test and thus, depending on the order of executions in the test suite, this test can pass or fail.

2.1.3 Categories

Several studies were conducted in an attempt to categorise flaky tests based on their root causes [49], [58], [60]–[62]. The categories slightly differ depending on the programming languages but the prevalent ones remain. Table 2.1 list the most common flakiness categories.

2.2 Machine Learning

Machine learning is a subfield of artificial intelligence focusing on the development of algorithms and models being able to learn and make predictions without being explicitly programmed. It involves the utilisation of statistical techniques and computational power to analyze and interpret large datasets, identifying patterns and relationships within the data. By iteratively learning from many examples and experiences, machine learning algorithms improve their performance and can generalize to make accurate predictions or take informed actions on new, unseen data [63], [64]. Machine learning can be divided into two subgroups: supervised learning and unsupervised learning.

2.2.1 Supervised Learning

Supervised learning is a machine learning approach relying on using labelled datasets. These datasets are collected and then used to train models, *i.e.* supervising them, to classify data or to predict outcomes accurately. Supervised learning problems can further be divided into two families:

Classification problems aim at predicting the particular group or category for a data item. Famous examples of this problem are the binary classification of emails (spam or non-spam), or multi-class classification of a given handwritten character (one class for each letter in the alphabet). Common algorithms used for classification tasks include logistic regressions, support vector machines, decision trees and random forests.

Regression problems aim at predicting a continuous numerical output variable based on input features. It involves building a regression model that can learn the relationship between the input variables (also known as independent variables, features, or predictors) and the output variable (also known as the dependent variable or target). Such an approach can be used to predict the price of houses based on features like living area and year of construction for example. Common algorithms used for regression tasks include linear regression, polynomial regression, decision tree regression, random forest regression, support vector regression, and neural network regression

2.2.2 Unsupervised Learning

Unsupervised learning is a machine learning approach that learns to discover inherent patterns or relationships in the data and extract meaningful insights or representations without the need for labelled datasets. Unsupervised learning can also be subdivided into two groups:

Clustering algorithms group similar data points together based on their features or characteristics. The algorithm recognises patterns in the data and assigns data points to different clusters. Examples of clustering algorithms include k-means clustering, hierarchical clustering, or DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [65], [66].

Dimension reduction algorithms aim to reduce the number of features in a dataset while preserving important information. These algorithms transform high-dimensional data into a lower-dimensional representation. Principal Component Analysis (PCA) and t-SNE (t-Distributed Stochastic Neighbor Embedding) are commonly used dimensionality reduction techniques [67], [68].

2.2.3 Performance Evaluation

Several chapters in this dissertation leverage supervised learning with binary or multi-class classification problems. To evaluate the performance of binary classification models, we rely on different metrics derived from true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). Precision measures the accuracy of positive predictions, while recall measures the completeness of positive predictions. In other words, precision measures how many detected items are relevant. It is calculated by dividing the true positives by the overall positive elements. Recall measures how many relevant elements were detected. Therefore it is calculated by dividing true positives by the number of relevant elements.

$$\textbf{Precision} = \frac{TP}{TP + FP} \qquad \textbf{Recall} = \frac{TP}{TP + FN}$$

In addition to Precision and Recall, the F-score or F1 score is also often given. It's an accuracy measure calculated as the harmonic mean between Precision and Recall.

$$\textbf{F1} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The accuracy of a model is sensitive to class imbalance. In particular, the precision and recall metrics can easily be impacted when one class is underrepresented. To alleviate this issue, we report the Matthews Correlation Coefficient (MCC) which is a more reliable statistical rate to avoid over-optimistic results in the case of an imbalanced dataset [69]. This metric takes into consideration all four entries of the confusion matrix. MCC ranges from -1 to 1 and is given by the following formula:

$$\textbf{MCC} = \frac{TN \times TP - FP \times FN}{\sqrt{(TN + FN)(TP + FP)(TN + FP)(FN + TP)}}$$

2.3 Spectrum-Based Fault Localisation

As explained in the introduction, Chapter 7 will present solutions based on SBFL to identify classes responsible for flakiness. This section introduces SBFL giving the necessary details to understand the contribution.

Line	Program	Tests		Susp. score
		t1	t2	
1	`int maximum(int a, int b) {`	■	■	0.5
2	`    if (a > b) {`	■	■	0.5
3	`        return b; //Fix: return a;`	■	□	1
4	`    }`	■	□	1
5	`    else {`	□	■	0
6	`        return b;`	□	■	0
7	`    }`	□	■	0
8	`}`	✘	✔	

Figure 2.3: SBFL example

Spectrum-based fault localisation is a technique that aims at identifying the locations of faults in a software program by analyzing the coverage-based spectrum by running the passing and failing tests. SBFL formulas are used to compute a suspiciousness score for each program component. Usually, a score of 1 is given to the most suspicious component (usually a program statement) and a score of 0 to the least. Figure 2.3 illustrates an example of a faulty program, the coverage information about two test cases, one failing and one passing, and the suspiciousness score derived from an SBFL formula. The bug lies on line 3 in the return statement. Test case T1 failed after covering the first 4 lines. Test case T2 passed without having covered lines 3 and 4. We see that most suspicious lines are actually lines 3 and 4.

2.3.1 Formulas

Several formulas have been introduced in the literature. The most common ones are Tarantula [70], Ochiai [71], DStar [72] and Barinel [73].

$$Tarantula : \mathbf{S(s)} = \frac{failed(s)/totalFailed}{failed(s)/totalFailed + passed(s)/totalPassed}$$

$$Ochiai : \mathbf{S(s)} = \frac{failed(s)}{\sqrt{totalFailed * (failed(s) + passed(s)}}$$

$$DStar : \mathbf{S(s)} = \frac{failed(s)^*}{passed(s) + (totalFailed - failed(s))}$$

$$Barinel : \mathbf{S(s)} = 1 - \frac{passed(s)}{passed(s) + failed(s)}$$

Where *s* denotes a statement in a program, *S(s)* represents the suspicious score computed for a given statement *s*, *passed(s)* and *failed(s)* are the number of, respectively, passed and fail executions for which the statement *s* was covered, and *totalFailed* and *totalPassed* are the total number of passing and failing executions.

Table 2.1: The different categories of flakiness commonly reported by the literature.

Category	**Definition**	**Sources**
Asynchronous Waits	Flakiness caused by tests that involve asynchronous operations and have dependencies on timing, resulting in inconsistent behaviour if the expected response is not received within a specified time.	[49], [58], [61], [62]
Concurrency	Flakiness caused by race conditions or synchronisation issues when multiple threads or processes interact with shared resources simultaneously, leading to unpredictable outcomes.	[49], [58], [61], [62]
Time	Tests depending on specific timing conditions, such as time-sensitive calculations or time-based events, and may produce different results based on the time of execution.	[49], [58], [61], [62]
Order-Dependency	Flakiness resulting from tests that rely on a specific execution order due to shared resources or dependencies, and may fail if the order among the tests is changed.	[49], [58], [61], [62]
Randomness	Flakiness caused by tests that involve random or pseudo-random behaviour, where different outcomes may occur on each run, potentially leading to inconsistent results.	[49], [58], [61], [62]
Unordered Collections	Flakiness resulting from tests that rely on unordered collections or sets, where the order of elements can vary, causing failures if the expected order is not maintained.	[58], [61], [62]
Network	Flakiness caused by network-related issues, such as unreliable connections, timeouts, or network congestion, leading to inconsistent results in tests that interact with remote services.	[49], [58], [61], [62]
I/O (Input/Output)	Flakiness resulting from tests that involve reading from or writing to external files, databases, or other I/O operations, where inconsistencies or errors can occur.	[49], [58], [61], [62]
Resource Leak	Flakiness caused by tests that do not release system resources properly, resulting in resource exhaustion and inconsistent behaviour when run repeatedly.	[49], [58], [61], [62]
Floating Point	Flakiness caused by tests that rely on the results of floating point operations, which can suffer from discrepancies and inaccuracies due to precision limitations, overflows, non-associative addition, and other factors.	[49], [58], [62]
Platform Dependency	Flakiness stemming from tests relying on specific functionalities of an operating system, library version, or hardware vendor. These dependencies can result in inconsistent and non-deterministic test failures, especially in cloud-based continuous integration environments where tests are executed on different platforms.	[49], [61]
Test Case Timeout	Flakiness caused by tests that specify an upper limit for the test execution duration. Often those tests will fail because the instructions will not complete in time.	[49], [61]

3 Related Work

This chapter presents the related work to the different contributions of this dissertation. It focuses on flakiness research: the main empirical studies and research tools available at the time of writing. It also covers the existing studies on flakiness root-causing and flakiness prediction.

Contents

3.1 Empirical Studies on Flakiness

Flakiness is a known issue in software testing but research studies have only gained momentum in the past few years [74].

The first study on test flakiness was carried out by Luo *et al.* [58]. They analysed 201 commits from 51 open-source projects in order to understand the root causes of flaky tests. They showed that Async Waits, Concurrency, and Test Order Dependency are the main categories of flakiness. Later studies replicated the work of Luo *et al.*, showing that other flakiness root causes can be more predominant in different application domains. Gruber *et al.* [61] presented a large empirical analysis of more than 20,000 Python projects. They found Test Order Dependency and Infrastructure to be among the top reasons for flakiness in those projects. Thorve *et al.* [75] analysed 77 flakiness-related commits in 29 open-source Android applications and found that 22% of these commits have flakiness caused by external factors like Hardware, Operating System version, and third-party libraries. Other studies also focused on particular software contexts [76]–[78] or on highlighting the effects of flakiness on other software testing techniques such as mutation testing and program repair [51], [79], [80].

Several studies have been conducted to inspect flakiness in the industry. Leong *et al.* [57] studied flaky tests at Google and found that more than 80% of test output transitions are caused by flakiness.

Eck *et al.* [49] surveyed 21 Mozilla developers, asking them to classify 200 flaky tests in terms of root causes and fixing efforts. This study highlighted four new categories of flakiness: restrictive ranges, test case timeout, test suite timeout, and platform dependency. It also provided evidence about flakiness from the CUT and showed that flaky tests can have organisational impacts. Other surveys were also conducted to replicate this one [60], [81] and Parry *et al.* conducted a wide literature review amounting 76 papers [74]. In our Chapter 4, we also conduct a survey but we leverage a qualitative approach (interviews) to address other aspects of flakiness in practice. More specifically, we investigate broader sources of flakiness (*e.g.* SUT and infrastructure) instead of the root causes (*e.g.* concurrency and timeouts). Our study also inspects the actions taken by practitioners in order to prevent, detect, and alleviate flaky tests. Result-wise, our findings confirm the observations of Eck *et al.* about (i) the impact of flakiness on the test suite reliability and (ii) the challenges of reproducing and debugging flaky tests. Furthermore, we highlight new flakiness impacts, on testing practices and product quality, and we synthesise a list of automation challenges for flakiness mitigation.

Durieux *et al.* [82] investigated the impact of flakiness and restarted builds on development workflow in Travis CI projects. They found that the more complex the project, the more likely it was to contain restarted builds. They also showed that those builds slowed down the merge of pull requests by a factor of three.

3.2 Flakiness Research Tools

To help debug, reproduce, and comprehend the causes of flaky tests several tools have been introduced. *DeFlaker* [83] detects flaky tests across commits, without performing any reruns, by checking for inconsistencies in test outcomes with regard to code coverage. Focused on test order dependencies, *iDFlakies* [59] detects flaky tests by rerunning test suites in various orders.

To increase the chances of observing flakiness, Silva *et al.* [84] introduced *Shaker*, a technique that relies on stress testing when rerunning potential flaky tests.

3.3 Root Causing Flakiness

One of the first main contributions to flakiness root cause localisation was proposed by Lam *et al.* [85]. They introduced a framework that helps developers at Microsoft to localise the root causes of their flaky tests. This framework uses an instrumentation tool to log the runtime properties of the test execution. Then it reruns the tests 100 times to produce logs for a passing and a failing execution. To analyse these logs and localise the root cause, they propose RootFinder, a tool that compares the logs of passing and failing executions to identify methods that can be responsible for flakiness. RootFinder relies on a predefined set of non-deterministic method calls and does not explore calls of unknown methods. Hence, it can only detect flaky tests that arise from method calls that the developer is already suspecting. In a second study, Lam *et al.* presented *FaTB*, an automated tool that speeds the runtime of test suites by lowering timeouts and waits without impacting the overall test suite flake rate. Romano *et al.* [76] analysed User Interface (UI) tests and showcased the flakiness root causes and the conditions needed to fix them.

Zitfci and Cavalcanti [86] presented Flakiness Debugger, a tool that compares the code coverage of passing and failing executions to localise the flakiness root cause. They ran their tool on 83 flaky tests and presented the localised root cause to two developers asking them for their evaluation. On average the developers found that in 48% of the cases, flakiness was due to the exact statements spotted by Flakiness Debugger. Moreover, only 18% of the outputs were considered inconclusive, hard to understand, or not useful. Both RootFinder and Flakiness Debugger relied on differences between passing and failing executions of flaky tests to localise flakiness in the CUT. In this study, we explore a new direction by analysing the differences between flaky and stable tests.

Following their work with iDFlakies, Shi *et al.* presented iFixFlakies [87] to automatically repair flaky tests. They introduced a framework that recommends patches for order-dependent flaky tests based on test patterns found in non-flaky tests that exhibit similar behaviour as flaky tests. Follow-up works were also conducted in this area [88].

Morán [89] presented FlakyLoc, a tool for localising the root causes of flakiness in web applications. The tool reruns web tests while varying environmental factors (network, memory, CPU, browser type, operating system, and screen resolution) and records test results. Then, it uses ranking metrics (Ochiai and Tarantula [70], [90]) to identify the environmental factor and values that are responsible for the flaky failure. The tool was only evaluated on one test case and it detected that the failure was caused by low screen resolution. In Chapter 7, we will also address flakiness localisation, but we do not focus on any specific flakiness category and our analysis is based on the test coverage instead of environmental factors.

3.4 Flakiness Prediction

Table 3.1: Existing machine learning-based studies aiming at detecting test flakiness. The majority of the techniques focus on detecting flaky tests, while half of the approaches rely on vocabulary features.

Study	**Model**	**Feature category**	**Features**	**Benchmark**	**Target**	**Year**
King et al. [91]	Bayesian network	Static & dynamic	Code metrics	Industrial	**Flaky tests**	2018
Pinto et al. [92]	Random forest	Static	**Vocabulary**	DeFlaker	**Flaky tests**	2020
Bertolino et al. [93]	KNN	Static	**Vocabulary**	DeFlaker	**Flaky tests**	2020
Haben et al. [94]	Random forest	Static	**Vocabulary**	DeFlaker	**Flaky tests**	2021
Camara et al. [95]	Random forest	Static	**Vocabulary**	iDFlakies	**Flaky tests**	2021
Alshammari et al. [96]	Random forest	Static & dynamic	Code metrics & Smells	FlakeFlagger	**Flaky tests**	2021
Fatima et al. [97]	Neural Network	Static	CodeBERT	FlakeFlagger iDFlakies	**Flaky tests**	2021
Pontillo et al. [98]	Logistic regression	Static	Code metrics & Smells	iDFlakies	**Flaky tests**	2021
Lampel et al. [99]	XGBoost	Static & dynamic	Job execution metrics	Industrial	Flaky failures	2021
Qin et al. [100]	Neural Network	Static	Dependency graph	Industrial	**Flaky tests**	2022
Olewicki et al. [101]	XGBoost	Static	**Vocabulary**	Industrial	Flaky builds	2022
Ackli et al. [102]	Siamese Networks	Static	CodeBERT	Various	**Flaky tests**	2022

While they remain scarce, the recent publication of datasets of flaky tests [59], [61], [83] enabled new lines of work. Prediction models were suggested to identify flaky tests from non-flaky tests. King *et al.* [91] presented an approach that leverages Bayesian networks to classify and predict flaky tests based on code metrics. Pinto *et al.* [92] used a bag of words representation of each test to train a model able to recognize flaky tests based on the vocabulary from the test code. This line of work has gained a lot of momentum lately as models achieved higher performances. Several works were carried out to replicate those studies and ensure their validity in different settings [95] including the work presented in Chapter 5.

In an industrial context, Kowalczyk *et al.* [103] implemented a flakiness scoring service at Apple. Their model quantifies the level of flakiness based on their historical flip rate and entropy (*i.e.* changes in test outcomes across different revisions). Their goal was to identify and rank flaky tests to monitor and detect trends in flakiness. They were able to reduce flakiness by 44% with less than 1% loss in fault detection. In the study conducted in Chapter 8, we also rely on test history to help with flakiness prediction.

More recently, *FlakeFlagger* [96] has been proposed. It builds a prediction model using an extended set of features from the code under test together with test smells. The research community continue to draw attention to this field by considering other possible features to predict flaky tests, this is the case of Peeler [100] for example, where the authors leveraged test dependency graphs to predict flaky tests.

Less attention has been given to flaky failures or false alerts prediction. Herzig *et al.* [104] used association rules to identify false alert patterns in the specific case of system and integration tests that contain steps. They evaluated their approach on Windows 8.1 and Microsoft Dynamics AX.

Olewicki *et al.* investigated the possibility of leveraging vocabulary-based features on logs from failing builds to predict if failures are caused by defects in the code or by other non-deterministic issues including flaky test failures. It is interesting to note that their work focuses on builds and not tests as we do in our study.

Finally, a recent study by Lampel *et al.* [99] presented an approach that automatically classifies jobs by deciding if a job failure originates from a bug in the code or from flakiness. To do so, they relied on features from job executions, *e.g.* CPU load, and run time. As such they are only concerned with some form of flaky failures and not with the utility of detecting flaky failures in CI, instead of tests as we do in this paper.

Table 3.1 summarizes the state-of-the-art of flakiness prediction in chronological order. By inspecting the table, we see that most of the studies focus on flaky tests, with just one focusing on flaky test failures. We also notice that static features like code metrics and test smells are often used but features based on vocabulary (*i.e.* bag-of-words) are the most popular ones. In Chapter 8, we will see why focusing on flaky failures instead of tests is important and how challenging it is to get good prediction performance.

4 A Qualitative Study on the Sources, Impacts and Mitigation Strategies of Flaky Tests

At the time of writing, flakiness research was still in its early stages. Therefore, in this chapter, we answer the need for a better understanding of the problem by conducting a qualitative study. We interviewed practitioners and report their perception of the sources and impacts of flakiness as well as the mitigation strategies they have in place to tackle the problem. We also identify automation opportunities paving the way for future research works.

This chapter is based on the work published in the following paper:

- S. Habchi, G. Haben, M. Papadakis, *et al.*, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 244–255. DOI: 10.1109/ICST53961.2022.00034

Contents

4.1 Introduction

Software Testing is critical for modern software development as it allows the concurrent implementation and integration of features. At Google, more than 50 million test cases are executed every day to ensure the quality of their products [106]. Though, test automation faces major problems with the emergence of flaky tests [53]–[55]. Flaky tests are tests that, for the same versions of code and test, can pass and fail on different runs. Such non-determinism sends confusing signals to developers who struggle to interpret the test results. As a result, developers lose trust in test suites, disregard their signals and integrate features containing real failures, thereby nullifying the purpose of testing.

Flaky tests are prevalent in large software systems and they incur significant costs. Google reports indicate that 16% of their tests exhibit some flakiness whereas 84% of the transitions from pass to fail involve a flaky test [55].

In response to this challenge, researchers dedicate their efforts in understanding the nature of flaky tests and the way they manifest. Empirical studies examined the root causes of flaky tests in open-source software [49], [58], [75], [107] and industrial systems [85], showing that concurrency and order-dependency are among the main categories of test flakiness. Notably, the study of Eck *et al.* [49] showed that flakiness can stem from the code under test and highlighted its potential impact on organisational aspects like resource allocation.

Other studies investigated tools and techniques that could help developers to cope with test flakiness. Automated tools, such as DeFlaker [83], iDFlakies [59], and FlakeFlagger [96] have been developed in order to detect flaky tests with a minimum number of test runs or re-runs. Unfortunately, these advances offer only partial solutions to the problem and may not fit well within the development systems and organisation constraints. For instance, DeFlaker relies on coverage and reruns of tests that do not execute changed code, which are not possible in specific development environments that use regression test selection or when coverage cannot be obtained. Furthermore, the fixing of flaky tests gained traction as studies investigated the fixing effort devoted for flaky tests and tools like [87] are designed to fix flaky tests. Nonetheless, in order to devise flakiness solutions, we need to understand how developers deal with flaky tests in practice. In particular, it is necessary to identify the typical measures taken by practitioners when dealing with flaky tests, and reflect on how research solutions could assist and improve them.

To shed some light on these questions, we conduct an empirical study focused on the industrial context in which flakiness manifests. Specifically, we perform a qualitative analysis on data collected from 14 practitioner interviews to answer the following research questions:

RQ1: *Where can we locate flakiness?*

Goal: Differently from previous studies [49], [58], [75], [85], [107], which focused on the root causes of flakiness, *e.g.* concurrency and timeouts, we aim to identify where flakiness stems within the different components of the development ecosystem, *e.g.* test, code under test, and infrastructure. This localisation is necessary to guide both detection and fixing approaches.
Results: In addition to tests, flakiness stems from the poor orchestration between the system components, the testing infrastructure, and external factors, *e.g.* OS and firmware. Studies should consider and leverage these factors when addressing flaky tests and not focus solely on the test and code under test.
RQ2: *How do practitioners perceive the impact of flakiness?*
Goal: This question is commonly discussed in industrial reports and research studies. In this paper, we examine it through direct discussions with practitioners. The aim is to understand the impact of flakiness on the development workflow and practices.
Results: Besides dissipating development time and hindering the continuous integration (CI), flakiness affects the testing practices and leads to a degradation of the system quality. We also shed light on the pernicious consequences of system flakiness, *i.e.* buggy or non-deterministic features that are falsely labelled as flaky tests.
RQ3: *How do practitioners address flaky tests?*
Goal: This question aims at identifying and understanding the measures taken by practitioners to address flakiness before and after it manifests in the CI.
Results: The prevention of test flakiness is performed by building stable infrastructures and enforcing guidelines, whereas the detection still relies mainly on reruns and manual inspection. Our results also highlight monitoring and logging tasks, which are commonly dismissed in research, yet they are key to most of the mitigation measures taken by practitioners.
RQ4: *How could mitigation measures be improved with automation tools?*
Goal: This question aims to identify specific needs to be addressed by future research.
Results: We accentuate the need for techniques that monitor and analyse the system states to assist the prediction, debugging, and fine-grained evaluation of flaky tests. Our participants also expressed the need for automating the quality assessment of software tests through static analysis and variability-aware reruns, *i.e.* reruns under diverse system configurations.

We believe that the qualitative results of this study are necessary to complement the current understanding of flakiness and advise future work.

4.2 Preliminary Analysis

We conduct a grey literature review (GLR) to establish an initial mapping of the measures adopted by practitioners when dealing with flaky tests. This mapping lays the foundation for our mitigation analysis (RQ3) and helps in guiding our interview design. With respect to this objective, this GLR is exploratory and non-exhaustive. In the following, we explain our process for collecting, evaluating, and analysing data from the grey literature.

4.2.1 Search

We followed the recommendations of Kitchenham and Charters [108], for the reviewing process in general, and the guidelines of Garousi *et al.* [109] for the aspects specific to grey literature. The research question for our review is:

RQ3: *How do practitioners address flaky tests?*

In order to answer this question, we focused our review on materials published by practitioners describing their mitigation of flakiness, *e.g.* technical reports, presentations, blogs, etc. To collect these materials, we queried the advanced Google search engine with the following string: `(Mitigate OR Manage OR Deal OR Control OR Avoid OR Prevent OR Tools OR Identify OR Detect) AND (Flaky OR Intermittent OR Unreliable OR non-deterministic) AND Tests`.

This query resulted in $276,000$ results. We manually checked the top 100 articles and only accepted articles that:

- Are written by practitioners. Articles and Blog posts written by researchers are excluded.
- Depict practitioners' views on flakiness and do not only address the problem theoretically.

We found that only 56 articles correspond to the searched material as a large part of the top-100 articles were dedicated to the introduction of flakiness without addressing its mitigation.

4.2.2 Analysis

The objective of this step is to identify and categorise the flakiness mitigation measures from the selected articles. For this purpose, we first examined the 56 articles to check their adequacy for our analysis. We relied on the quality assessment checklist presented by Garousi *et al.* [109], which is specifically designed for grey literature sources. We found that three factors are particularly relevant in our context and we adopted them as exclusion criteria:

Objectivity We exclude sources where the authors have a clear vested interest. For instance, articles that promote new tools or plugins for mitigating flaky tests are generally biased.

Method adequacy We found that very few sources have a clearly stated their aim and methodology. However, from the presented content, we could identify articles that were not based on practical experience and exclude them. For instance, in several cases, the authors present mitigation measures from a compilation of other sources and not based on their own experiences.

Topic adequacy We checked whether the articles enrich our analysis or not. More specifically, we excluded articles that do not present any mitigation measures for flaky tests.

The full quality assessment is available with our artefacts [110]. Based on the three exclusion criteria, we selected 38 articles that fit within the study scope and objectives. Two authors read these articles and iteratively synthesised a classification of the measures described by practitioners. This consensual process is similar to the qualitative analysis performed on the interview transcripts (*cf.* Section 4.3). The results of this analysis are presented in Table 4.2 and will be discussed in Section 4.4. Interestingly, in our grey literature analysis, we observed that the articles do not explain the rationale behind the choice of measures. Similarly, the consequences of the measures are generally dismissed. Hence, we try to address these gaps in our interviewing process.

4.3 Interviews and Analysis

The objective of the interviewing process is to explore the topics of our research questions with an open mind instead of testing pre-designed questions. For this purpose, we pursue a qualitative research approach [111] based on classic Grounded Theory concepts [112]. In this section, we explain our implementation of this approach from the interview design to the analysis of the results.

4.3.1 Questions

Since we already formulated our topics of interest (RQs), we opted for semi-structured interviews. These interviews build on starter questions, which cover the topics of interest, and according to the interviewee's answers, they develop follow-up questions that explore other points. While designing and conducting our interviews, we followed the recommendations of Hove *et al.* [113]. In particular, we ensured the clarity of the discussed topics and notions before going through the interviews. For instance, we always asked questions about the interviewee's definition of flakiness to avoid misunderstandings and ensure that the following questions are interpreted correctly. We also avoided making prior assumptions about participants' opinions or actions. For example, we ask several questions about the testing practices before formulating our questions to avoid wrong assumptions about the use of automated testing or CI. We also explained the non-judgemental nature of the interviews and encouraged participants to express their opinions freely. Specifically, we mentioned that the objective is not to assess the participants' knowledge about flakiness but rather to grasp their perception of it. Finally, we asked follow-up questions whenever possible and we favoured open questions such as *"Why did you opt for this measure?"* to incite participants to explain their motivations. We structured our interview around the three following sections.

Context We asked questions to characterise the project and testing infrastructure.

1. What kind of projects do you work on? If possible ask for metrics like codebase size, architecture, and development team size.
2. Do you have automated or manual tests?
3. What kind of tests do you generally write?
4. Do you have a continuous integration?
5. Do you have a testing policy?
6. Can you describe your testing infrastructure? Do you consider it stable?

Flakiness We asked general questions about flakiness:

7. Do you know what a flaky test is?
8. What is your definition of flakiness?
9. How commonly do you encounter flaky tests?
10. What are the sources of flakiness in your context?
11. Do you consider flakiness as an issue? Why?

Measures We asked questions about the actions taken by participants to prevent and address flaky tests:

12. How do flaky tests manifest in your codebase? How do you detect them?
13. How do you treat the identified flaky tests?
14. Do you adopt any specific measures to avoid flaky tests?
15. Why did you adopt these measures?
16. Do you face difficulties when dealing with flaky tests?
17. If yes: What are these difficulties and what could help you to overcome them?

For each measure described by the participant, we asked follow-up questions to understand the motivations and consequences. When possible, we also asked follow-up questions about the measures that the participants did not take, *e.g.* if they never mention fixing flaky tests, we could ask about the rationale behind it. All the interviews were performed with online calls where we explicitly asked the participants for recording permission. The recordings lasted from 26 to 63 minutes with an average duration of 41 minutes.

Participants

Our objective was to select practitioners who have experience in dealing with flaky tests in diverse contexts. This diversity enriches the study and allows us to have a thorough understanding of the practitioner perspective. To ensure this diversity, we relied on several channels to invite potential participants.

We shared our invitation with a brief description of the study objectives on online groups for software engineering practitioners. For instance, we targeted a group that gathers 265 practitioners that are interested in software testing and continuous integration. The group members are from large companies like Tesla, Google, Apple, VMWare, Netflix, Facebook, Spotify, etc. To include participants from other backgrounds, we also targeted groups of practitioners from FinTech companies, average-sized IT companies, and local startups. Following these invitations, we received answers from 19 practitioners who showed an interest in our study. After exchanges, five participants estimated that their experience is insufficient for the study and did not proceed with the interviews, thus our process ended up with 14 participants. This number of interviewees is typical in studies that approach similar topics [114], [115]. Besides, due to the specificity of the topic, it is very challenging to find other developers that are qualified enough to take part in the study. We conducted the interviews with the 14 participants and after the analysis, we considered that the collected data is enough to answer our research questions and provide us with theoretical saturation [116]. Indeed, the three last interviews did not lead to any changes in our analytical template and only provided new formulations for existing categories.

Table 4.1 summarises the profile of our participants (role and years of experience) and their current companies (number of employees, domain, and number of users). To preserve the anonymity of our participants, we refer to them with code names, we omit their company names, and upon specific request, we also omit the experience and domain. Our participants have solid experience in software engineering, their experience ranges from 6 to 35 years, with an average of 16 years. The participants also work in companies that vary significantly in terms of size and domain of activity. On top of the industrial experience, three of our participants contributed regularly to Open Source Software (OSS) as part of their job or as a side activity.

4.3.2 Analysis

As our study builds on semi-structured interviews, we relied on the strategy proposed by Schmidt *et al.* [117]. This strategy helps with inquiries where a prior understanding of the problem is postulated but the analysis remains open for exploring new topics and formulations. In the following, we explain the four steps of this analysis.

Transcription To prepare the interview analysis, we transcribed the recorded interviews into texts following a denaturalism approach. This approach allows us to dismiss non-informational content and ensures a full and faithful transcription [118]. For the cases where the interviews were not conducted in English, we transcribed them in the original language and we only proceeded to their translation at the reporting step.

Table 4.1: A summary of participants' profiles.

	Role	Years	Size	Domain	Users	OSS
P1	Engineering Manager	24	+1K	Music	+200M	No
P2	CTO	10	+10	Mobility	-	No
P3	Tech Lead	7	+200	Cloud	+30K	Yes
P4	QA Consultant	12	+2K	FinTech	+190K	No
P5	CTO	14	+10	Infrastructure	-	No
P6	Staff Engineer	20	+1K	DevOPs	-	Yes
P7	Vice President	17	+200	Cloud	+30K	Yes
P8	Architect	7	+5k	Online sales	+70M	No
P9	Senior Researcher	35	+20	R&D	-	No
P10	Architect	30	+24K	Virtualisation	+500K	No
P11	Senior Engineer	6	+10k	-	+500M	No
P12	Principal Architect	23	+10k	Payment	+200M	No
P13	Front-end Developer	7	+40	Banking	-	No
P14	Senior Engineer	-	+10k	-	+500M	No

Definition of analytical categories The goal of this step is to define the analytical categories that guide our analysis. In our case the initial categories of interest were (i) *the sources of flaky tests*, (ii) *the measures for mitigating flaky tests*, and (iii) *the difficulties of dealing with flaky tests.* After conducting four interviews, we observed that an additional topic that is commonly mentioned by developers is: (iv) *the impact of flakiness.* Based on our preliminary discussions, this topic provided new insights on the effects of flaky tests, as seen by practitioners. This topic also seemed essential for understanding the efforts dedicated to the mitigation of flaky tests. Hence, we added this topic to our categories of interest and our interview template. After setting the analytical categories, we read each participant answer to identify the categories that can be associated with it. In this process, we do not only focus on the participants' direct answers, but we also consider their use of terms and the aspects that they omit. For instance, in our analysis of the second analytical category, we consider the measures taken by practitioners but also those that they were not aware of or the ones that they discarded. On top of that, we carefully analyse developers' answers to context and flakiness questions to spot elements that can help in interpreting their answers.

Assembly of a coding guide The objective of this step is to build a guide that can be used to code the interviews. We assembled the four analytical categories and identified different sub-categories for them based on an initial reading of the interviews. The sub-categories represent different versions formulated by developers in one analytical category. For instance, for the first analytical category, *i.e.* sources of flaky tests, the initial sub-categories were Test, Code Under Test, and Environment. These sub-categories are not final and can be refined, omitted, or merged along the following step. For example, the sub-category Environment is later refined to two categories Infrastructure and Uncontrollable environment.

Coding We read the interviews to identify passages that can be related to the categories and sub-categories of our coding guide. This process can be repetitive as every time a new sub-category is identified or refined, we need to read previous texts to ensure that all passages related to it are identified. To ensure the soundness of this process, two authors coded the interviews separately before comparing their results. In case of disagreement, the authors discussed their views and opted for a negotiated solution. Besides this consensual coding, all the authors discussed the coding guide iteratively, to ensure the clarity and precision of the identified sub-categories.

4.4 Results

4.4.1 RQ1: Where can we locate flakiness?

Test

8 participants mentioned that the test itself, when poorly written, is a cause of flakiness (P1, P2, P3, P4, P6, P8, P13, P14). In particular, the participants explained that some tests are by nature difficult to write and prone to flakiness. For instance, GUI tests were considered as a special cause of flakiness by many participants. *"The synchronisation points in GUI tests are a major cause of flakiness... We wait for some elements of the web page (e.g. button) to proceed to the testing but some other elements could be necessary and lead the test to fail"* (P4). According to participants, other cases where it is difficult to write flakiness-free tests included time manipulation, threads, statistics, and performance tests. P8 described examples of tests that encode variables and properties that are not really useful for the test case and lead to non-deterministic behaviour. These variables could be related to the system, environment, or time and they can be avoided inside the test code.

Code Under Test (CUT)

In this sub-category, we consider flakiness that stems from the part of the system that is directly under test. Surprisingly, only 3 participants mentioned that their flakiness stems from the CUT (P1, P3, P7). The root causes of CUT flakiness are similar to the causes of test flakiness, as examples, the participants mentioned concurrency and time handling. Interestingly, flakiness in the CUT can have direct impacts on the product reliability and thus developers tend to take it more seriously. *"If the product itself is flaky, which is happening quite often, then* *you have got a problem because you actually publish code which is flaky, it breaks one out of three times"* (P1).

System Under Test (SUT)

This source of flakiness was mentioned by 9 participants (P2, P3, P4, P5, P6, P7, P8, P12, P14). Differently from the CUT, this sub-category considers the system as a whole and not only the part under test. The SUT emerges as a source of flakiness in complex systems where integration tests flake due to failing orchestration between the system components. *"It only takes one timeout in the communication between two services or other middleware like databases to make a test fail randomly"* (P2). The failing interactions can be a result of a misunderstanding of the system architecture and its impact on tests, *"the principle behind micro-services is that every service can fail, so we need to keep that in mind when writing integration tests"* (P2). The organisational structure can also add to the difficulty of writing stable integration tests as components can be maintained by distinct teams that do not communicate properly. *"Every team has the impression of working in a sandbox, they would rebase the production or generate a new sequential number and the tests of other teams will flake because of that"* (P8). Ideally, these dependencies should be documented or formalised and integration tests should account for them. Yet, P8 confirms that despite the recurrence of such incidents, developers remain reluctant to invest in their documentation.

Infrastructure

The testing infrastructure is the set of processes that support the testing activity and ensure its stability. 8 participants considered that their tests were flaky because of an unstable or improper testing infrastructure (P1, P4, P5, P6, P10, P11, P12, P14). For instance, P5 explained that most of their flaky tests were caused by a lack of resources, *"the test is getting throttled because we do not have enough CPU or memory quota for our database"*. P12 showed how flaky tests can emerge from a mismatch between the product design and its usage in the testing infrastructure, *"a single data source that would, in production, be used by only one user, now is used by several tests that may override each other's data"*. When flaky tests are caused by poor infrastructure, participants express more struggle in detecting and fixing them as the search space is broader and programmers are not always qualified for these tasks, *"CI issues are not like race condition where we can have a clear solution for it, this is difficult because it can be different things"* (P6).

Environment

11 participants explained that tests can flake because of external factors (P1, P2, P5, P6, P7, P8, P9, P10, P11, P13, P14). This source of flakiness differs from the infrastructure by considering all factors that developers cannot or should

not control. One common example of the environment is the hardware on which developers have almost no control, *"sometimes one batch of RAM sticks has an unidentified problem and the test is failing because of it"* (P7). The underlying Operating System (OS) is also subject to various changes that make it unpredictable and therefore a potential source of flakiness. One example of such cases is given by P1: *"if we test the app on devices, then we rely on some iPhone being up and if it decides to upgrade its OS at the exact same time then we have a problem"*. On top of the OS, tests can always be impacted by cumulative states of the machine that developers do not account for, *e.g.* firmware versions, memory state, and access to the internet.

The impact of the environment is particularly perceptible on GUI tests since they run on different web browsers that are prone to frequent changes. Similarly, developers may need to write acceptance and integration tests that depend on external resources that are hardly controllable. *"I work on a command-line interface that wraps packages from different providers, it seems simple but there are always random changes"* (P7).

It is worth noting that the distinction between infrastructure and environment may depend on the software, test type, and the choices of the practitioner. Some developers can consider aspects like the OS state as part of their infrastructure and control it to ensure the reliability of their tests, whereas others choose to ignore it. Likewise, aspects that seem external and futile for unit or integration tests, *e.g.* firmware, must be considered and controlled as part of the infrastructure of performance tests.

Testing framework

Two developers found that the testing framework can lead to flakiness (P1, P7). This issue can arise when the framework is written or customised by the developers themselves, which makes it less stable than other widely used frameworks. Another possible issue is the mismatch between the testing framework and the CUT. This can occur when the framework is not adapted to the type of tests or to the application domain. P7 describes a similar case: *"We used a Cassandra cluster (NoSQL) and we tried to test the database consistency rules. This generated many flaky tests. Instead, we should have used a more delicate testing framework to write serialisation tests and produce consistency edge cases"*.

Tester

Two participants believed that developers and testers can constitute a source of flakiness (P4, P5). This is possible for manual tests where the tester actions are part of the test execution. Indeed, being manual makes tests rely on human behaviour, which is less deterministic and more failure-prone. Hence manual tests can flake because of variations in the tester actions. Besides, the tester's misunderstanding of the requirements and the SUT can be another point of failure. *"The person running the tests does not always have a correct and precise idea of the behaviour expected from the system and this affects the test outcome"* (P4).

Discussion According to our participants, flaky tests stem frequently from the external factors of the environment, the interactions of the SUT, and the testing infrastructure. Flakiness is not limited to the test and CUT and the studies on this topic should consider and leverage all these factors when addressing flaky tests. Our analysis also shows that besides the well-established root causes of flaky tests, *e.g.* concurrency and order-dependency, the size and scope of the test are important flakiness factors. GUI and system tests are more prone to flakiness, yet, our understanding of flakiness in these types of tests remains limited and we still lack techniques that adapt to these specific tests.

4.4.2 RQ2: How do practitioners perceive the impact of flakiness?

It wastes developers' time

10 participants considered that flaky tests waste developers' time (P2, P4, P5, P6, P7, P8, P9, P11, P12, P14). When developers observe flaky failures, they have to invest time and effort in investigating the root cause before realising that it is a false alert. Besides the time wasted on investigating false alerts, our participants affirmed that discussions about flaky tests are also costly. *"It was ok when we were a team of five and everyone knew that the test is flaky. But as the startup grew, it became expensive and we found ourselves constantly explaining to other developers that these are not real failures"* (P7).

It disrupts the CI

7 participants mentioned that their flaky tests disrupt the continuous integration process (P2, P3, P4, P8, P10, P11, P13). This impact arises from the pace of modern development life cycles and its extent is proportional to the releasing frequency. *"Flakiness would never be an issue if we released once every two weeks. But in a CI today with 400 deliveries per day, disruptions waste so much time"* (P2). Disruptions also affect the developer's ability to develop confidently because the CI, which is supposed to guard the code quality, is halted, *"five days a month, the Jenkins of this project was red so I couldn't develop on the project and be sure that my work is not breaking anything at the time"* (P3).

It affects testing practices

6 participants observed that flakiness affects the testing practices in their teams (P1, P6, P7, P8, P12, P13). In particular, they explained how developers lost confidence in their capacity to write tests, According to P12, in the worst-case

scenario, developers are repelled and would write fewer tests to have fewer problems. In a phenomenon similar to the broken window theory, P1, P7, and P12 described how developers are more inclined to introduce and accept flaky tests in a system that is already flaky. *"As the suite is unreliable, it opens the door for more flaky tests"* (P7). Ultimately, the accrual of flaky tests pushes development teams to adapt their testing strategies: *"flakiness tends to accumulate in the system, and at some point, it becomes so large that companies may look for completely different solutions, like using more unit testing"* (P12). The impact on testing practices is not only related to flakiness but also to the general software quality, *"the more flakiness it is, the greater the acceptance of less than ideal test coverage, and that leads to a degradation of the software quality"* (P12).

It undermines the system reliability

5 participants highlighted the impact of flaky tests on the reliability of both tests and the SUT (P1, P3, P6, P7, P8). The false alerts raised by flaky tests confuse developers and make them question the suite's ability to detect faults accurately. Consequently, developers can disregard test results, which may lead to the introduction of bugs, *"if you do not fix flaky tests, people will start ignoring them and then they will introduce real bugs in the product"* (P1). Similarly, the non-deterministic test outcomes cast doubts on the reliability of the system under test. This doubt is all the more important in open source projects where newcomers can be repelled by inexplicable flaky failures. P6 who worked on a large open-source project stated: *"new contributors see CI failures, they do not know it is flakiness and it gives them the impression that the project is not well maintained so they do not even rerun the tests, they just give up"*.

It disguises bugs

Two developers explained that flakiness can hide buggy features (P1, P6). In some cases, the non-deterministic behaviour stems from a bug in the product, but as developers believe it is a flaky test, they disregard it without further inspection. *"People ignore the flaky test results because it is just a flake, except it is an actual problem in a product"* (P1). Interestingly, we witnessed first-hand the confusion between buggy features and flaky tests while performing the interview with P9. The participant was providing an example of non-deterministic test failures that were caused by memory issues, and when asked about how these flaky tests were detected, she replied: *"they appear when they are in the customer premises"*. After the customer complaint, the participant reran the test that covers the buggy code multiple times and reproduced the bug. In this case, the test has indeed a non-deterministic outcome, but addressing it as a flaky test (false alert) is inappropriate because the failure is real. Furthermore, the more flakiness is prevalent in a test suite the more developers are inclined to overlook non-deterministic system failures.

"The most important is that it actually makes people think they can introduce bugs in the form of flaky bugs in a product and get away with it" (P1).

Discussion Our results confirm the impact of flakiness in terms of development time and CI obstruction. Moreover, our analysis shows that the accrual of flaky tests affects the testing practices negatively as developers become repelled by testing and more lenient toward testing standards, which eventually leads to a degradation of the system quality. Our participants also raised the issue of system buggy non-deterministic features that are falsely labelled as test flakiness and therefore disregarded and shipped to end-users. For future studies, this shows the necessity of distinguishing the sources of flakiness and addressing them accordingly.

4.4.3 RQ3: How do practitioners address flaky tests?

Table 4.2 summarises the measures identified in our GLR. The columns *#GL* and *#Int.* report the number of times where the measure was mentioned in grey literature and interviews, respectively, while the columns *%GL* and *%Int.* report the percentages. The full results summary is available within our artefacts [110].

Prevention measures

This represents all proactive practices that aim to prevent the introduction of test flakiness.

Set up a reliable infrastructure Grey literature articles that embraced prevention measures estimated that a proper setup of the testing infrastructure is necessary for avoiding flaky tests. Several practitioners adopted hermetic servers, *a.k.a.* mock servers, where tests can be run locally without the need to call external servers [119]–[121]. Some articles also stressed the importance of using containers to ensure that the testing environment is clean when the tests are run [122]. 9 interviewees reported the adoption of similar practices to ensure the stability of their infrastructure (P1, P3, P4, P5, P6, P10, P11, P13, P14). P6 explained that they rarely observe test failures caused by infrastructure or environment thanks to their use of virtual machines. *"The virtual machine is started for the tests and destroyed just after ... all our tests are reproducible"* (P6). P4 mentioned *pre-tests*, a form of sanity checks, as another solution to infrastructure flakiness. *"If we have 5 APIs involved, the pre-tests check that these APIs are up, otherwise the test is not run"* (P4).

Define testing guidelines One guideline that was recurrently mentioned is following the testing pyramid principles [122]–[124]. These basic principles force developers to respect the scope of each test type and avoid flakiness. The proportions of each test type shall also be respected to avoid the *Ice cream cone* and the *Cupcake* anti-patterns [125], where the number of GUI tests, which are a main source of flakiness, is exaggerated. Interestingly, only two of our interviewees (P11 & P14)

Table 4.2: The number and percentage of grey literature articles and interviews for each mitigation measure.

	Strategy	#GL	%GL	#Int.	%Int.
Prevent	**Setup a reliable infrastructure** with processes properly adapted to testing activities.	4	11%	9	64%
	Define guidelines that should be respected when writing tests and enforced through reviews.	5	13%	9	64%
	Limit external dependencies by mocking dependencies.	9	24%	1	7%
	Customise the testing framework to avoid flaky features.	4	11%	1	7%
Detect	**Rerun** the failing test multiple times to check if it is a real or a flaky failure.	14	37%	7	50%
	Manually analyse the failure message and trace to determine if the test is flaky.	17	45%	3	21%
	Check the test execution history to distinguish flaky from real failures.	8	21%	2	14%
	Proactively expose test flakiness before it manifests in the CI.	5	13%	2	14%
	Compare test coverage to the modifications of the commit under test to identify flaky failures.	2	5%	1	7%
Treat	**Fix** the root cause of flakiness to remove the non-deterministic behaviour.	15	39%	7	50%
	Ignore flaky tests that are not common or costly (based on the flake rate & periodicity).	2	5%	5	36%
	Quarantine flaky tests by isolating them from the blocking path that commands the CI.	7	18%	4	23%
	Remove the test permanently.	2	5%	4	23%
	Document flaky tests in databases, issues, alerts, or internal reports.	8	21%	3	21%
Support	**Monitor and log** system interactions and test outcomes.	8	21%	9	64%
	Establish testing workflows that protect the CI.	2	5%	4	23%

confirmed that their teams defined explicit guidelines to prevent flakiness. P14 considered that thanks to these explicit guidelines, she rarely encounters flaky tests in her product, *"with the investment that was done in the guidelines and tooling, now we are able to cope with flakiness"*. The other participants suggested that the absence of explicit guidelines in their companies is due to the lack of maturity (P8). However, many participants affirmed that with experience their teams had developed testing practices to avoid flaky tests (P2, P3, P4, P6, P7, P10, P13). These practices are similar to the ones identified from the grey literature. They focus on the test scope and size and they address common flakiness sources like concurrency and time manipulations. In order to enforce these good practices, the participants relied on code reviews.

Limit external dependencies This practice is more relevant for unit tests, which are supposed to test narrow parts of the systems, than integration or GUI tests, which have to interact with other components. The analysed articles explain that some practitioners keep useless dependencies in their unit tests, which lead to flakiness [119], [124]. P3 mentioned that in order to avoid environment flakiness, her team tries to mock external services, use test doubles, and prefer in-memory resources (*e.g.* database and file system).

Customise the testing framework Sudarshan *et al.* [126] explained how they built their own testing framework so they can test critical aspects like time and concurrency without introducing flakiness. In some cases, practitioners customise the testing framework to disable features like animations in web and mobile applications, which are commonly connected to flaky tests [119].

Detection measures

This category groups all actions taken by developers to identify flaky tests.

Rerun Based on our GLR, reruns are the most common and intuitive way of identifying flaky tests despite their computation cost. Even other measures and mitigation steps, *e.g.* debugging and reproduction, require multiple test reruns. To maximise their chances to observe flakiness and minimise the number of reruns, the reruns can be performed in different environments (local machine, CI, etc) and with different settings (P4). Some participants advocated the effectiveness of reruns especially for infrastructure and environment flakiness (P1, P2, P4, P5, P10, P11, P12). Nevertheless, P1 warned about the consequences of solely depending on reruns to deal with flakiness, *"with reruns, you do not understand the issue and you can ignore actual problems"*.

Manually analyse test outcome When even reruns are not possible or useful, developers manually analyse the execution trace to determine if the test is flaky or not [127]. In the case of GUI tests, practitioners rely particularly on the screenshots recorded during the test run [120], [128], [129]. P2, P4, and P8 affirmed that they prefer going through manual analysis before trying reruns or other detection techniques. In the case of P8, this choice is due to system specifications that make rerunning the same test in the exact same conditions impossible.

Check test history Some practitioners keep a record of the test execution history, *i.e.* all test passes and fails for each build. When a suspicious test failure is observed, developers inspect these records to check if the test has already shown a random behaviour. Palmer *et al.* [54] argue that when these records are visualised they can help developers in distinguishing flaky failures easily and thus gain a lot of investigation time. P11 and P14 described a system in their company, which relies on the execution records to score tests. Based on the past passes and failures, a test receives a flakiness score that expresses the probability for this test to be flaky. P14 described how these scores helped her when a flaky test manifested, *"it is very good when it tells that it is 90% flaky and you can just go on with your day knowing that it's because of flakiness"*.

Expose As explained in RQ2, when a flaky failure occurs in the CI, it disrupts the work progress and wastes developers' time and efforts. For these reasons, some practitioners attempt to reveal flaky tests before CI failures [54], [130]. In this case, new tests are rerun several times to ensure that they are stable, before adding them to the main test suite. Among our interviewees, only P1 and P4 reported adopting this practice in their companies. *"Before committing the test, you should run it a thousand times (counting different configurations and device types) and it must be a thousand greens (passes)"* (P1).

Leverage test coverage When practitioners suspect that a test failure is flaky, they compare the coverage of the failing test to the modifications performed by the commit that triggered the build. If the intersection between these two is empty, the test is considered flaky. This process can be performed manually by developers (P14) or automatically using tools like DeFlaker [83]. However, P14 explains that, due to hidden dependencies between projects, this technique is not always effective.

Treatment measures

This presents actions taken by practitioners to deal with flay tests that manifested.

Fix In theory, every identified flaky test should be fixed at some point. However, according to practitioners, this point is rarely reached because the fix depends on two challenging steps, reproducing the flaky failure and determining its root cause (*cf.* RQ4). For this reason, many flaky tests remain unaddressed or removed. Interestingly, some participants affirmed that fixing flaky tests is easy when the root cause is known (P2, P10). P3 also affirmed that once the flaky test is understood, it was only a matter of resources to fix it.

Ignore Naturally, ignoring flaky tests is not commonly recommended in the grey literature (only 2 articles). Yet, 5 interviewees recalled situations where flaky tests were intentionally left unaddressed (P2, P3, P6, P7, P10). For P3 and P7, this was in a case where all team members were aware of the test flakiness and considered that the test is useful, so they did not isolate or remove it, but did not have enough time or resources to fix it. For P6 and P10, this choice is motivated by the severity of the flaky test, *i.e.* the flake rate. *"If the test has a very low flake rate, it is not really worth the investigation"* (P10).

Quarantine According to our GLR, quarantining flaky tests is one of the most common measures among practitioners. While in most cases, the isolation in quarantine is performed manually by developers when they identify a test as flaky, in some cases this process is more sophisticated. An article from Fuchsia explained how they designed an automated workflow where flaky tests are automatically identified and removed from the commit queue [131]. This workflow comprises a benchmark that evaluates the fixed flaky tests before reinserting them in the integration suite. By lack of better solutions, this evaluation relies on reruns. The adoption of the quarantine is less popular among our interviewees (P1, P4, P7, P10). Indeed, even participants who affirmed that they isolated their flaky tests, raised several questions about the side effects of this practice. P1 suggested that developers can abuse this practice, *"it's a dangerous way to go because then suddenly the number of tests goes down"*. P6 went further and considered that the quarantine is a bad practice because it implies that a potential bug is being disregarded without further investigation. *"You move the problem from the developer, who will not see the flaky failures anymore, and you transfer it to the user who may deal with a bug"* (P6).

Remove When a flaky test is hard to reproduce, debug, or fix, many practitioners recommend to remove it completely from the system to avoid its negative effects [131]–[133]. P1, P2, P7, and P14 affirmed that if a flaky persists and they are unable to address they choose to remove it. *"I would rather remove the flaky test from the codebase because of its cost"* (P2).

Document The documentation of flaky tests is performed for different purposes. The most basic being informing other developers that the test is flaky so they know how to react to its failures. The documentation is also helpful for the reproduction and debugging of flaky tests as it keeps logs, memory dumps, system states, screenshots in GUI tests, etc [134]. Finally, keeping track of all flaky tests is helpful when building a system that relies on execution history to detect flaky failures. Indeed, three interviewees affirmed that their internal systems relied on flaky tests that were documented in the past (P10, P11, P14) to guide developers when a test fails.

Support measures

This includes actions that are likely unrelated to test flakiness but are critical for addressing flaky tests.

Monitor and log 9 interviewees explained that when addressing a flaky test they rely mainly on the data logged by their monitoring system (P1, P6, P8-P14). P1 explained their advanced log analysis, which automatically suggests the root cause of the failure, *"we have a probe that can identify those root causes of flakiness"*. Regarding, the effect of this monitoring and analysis on their productivity, P1 added: *"it takes years to do it right, but it is extremely powerful"*. P11 and P14 explained that the test logs assist their flakiness prediction system. Furthermore, P6 and P10 showcased the importance of monitoring by affirming that their decisions are always guided by the flake rate, a test score that is calculated by monitoring and analysing test outcomes for periods of time.

Establish testing workflows For complex software systems, practitioners can design advanced testing paths that organise tests based on their criticality for the integration [129], [135]. In these scenarios, due to computation costs, the blocking path, *i.e.* the set of tests that decide in the CI, does not include all tests. 4 interviewees suggested that these workflows can be leveraged to protect the blocking path from flaky tests (P1, P10, P11, P14).

Discussion Our analysis shows that on top of the typical detection and treatment measures, developers take actions to prevent the introduction and manifestation of flaky tests. Interestingly, this prevention relies mainly on the setup of the infrastructure and the establishment of guidelines. To the best of our knowledge, these two tasks were not identified by prior studies and none of the literature techniques supports them. Similarly, our results emphasise the role of supporting measures like logging and monitoring in the accomplishment of critical mitigation steps like detection and fixing. The study of Lam *et al.* [85] has already shown that logs can be used to automatically spot the root cause of flakiness. Other studies should follow the same path and benefit from monitoring and log analysis to improve flakiness detection and prediction.

4.4.4 RQ4: How could mitigation measures be improved with automation tools?

Root cause identification and reproduction

8 participants expressed their struggle while reproducing and debugging flaky tests (P1, P2, P3, P4, P7, P9, P10, P11). These two tasks are tightly coupled because reproducing a flaky failure generally requires a minimal understanding

of the root cause. P4 explained that the difficulty of these tasks is due to the multitude and variety of potential factors of flaky tests, both in terms of root causes and sources (from the test itself to complete external factors). P11 added that the broadness of factors is particularly relevant for SUT flakiness: *"Trying to figure out among 8 to 10 services what is the actual culprit of flakiness is the challenging part"*. For all the participants, except P1, the reproduction and debug are currently performed manually, which is time and effort consuming. P7 affirmed that simple reruns are not always effective for reproducing and more advanced solutions are necessary, *"we need tracking tools to help us reproduce flaky tests"*. In the same vein, P4 said that even when logs are available, a lot of assistance is still required to help developers isolate the root cause and reproduce flaky tests.

Monitoring and log analysis

7 participants suggested that managing flaky tests would be easier if they were equipped with tools to monitor the testing activity and analyse the generated logs (P3, P4, P6, P8, P9, P12, P13). These two tasks are coupled because an automated analysis is critical to benefit from the data collected by the monitoring process. Indeed, P4 said that their GUI testing system produces overwhelming amounts of logs and yet it is impossible to manually draw insightful information from them. The analysis of such data can help developers to:

Predict flaky tests: As shown in RQ3, analysing the logs of test history is useful for predicting flakiness and assisting developers when a flaky failure occurs.

Identify the source or root cause: *"For debugging GUI tests, traces of all the called APIs can help in isolating the root of failure"* (P4).

Evaluate the flake rate: In RQ3, we showed that the flake rate monitoring gives a fine grained assessment of flaky tests and therefore guides the mitigation strategies, *e.g.* ignore flaky tests that flake rarely. *"This monitoring would help us to debug and find the changes that led to increasing the flake rate"*, stated P6 who explained that these tasks are currently performed manually.

Test validation

RQ3 showed that following testing guidelines is a key measure for preventing test flakiness. Yet, according to 9 participants, the process of enforcing these guidelines still relies on manual reviews, and it could be assisted with:

Static analysis: P10 described how preventing flakiness through code reviews can be redundant, *"I keep rejecting tests that have sleep() statements"*, and suggested that a simple static analyser could help in this regard. P4 described a similar situation with GUI testing reviews and affirmed that *"advanced static analysis could help to identify potential problems"*.

Variability-aware reruns: P4 mentioned that she currently tests the scripts of GUI tests manually: *"I test the script by crashing the browser and observing the* *outcome. This avoids pushing flaky tests that block the quality gate"*. P6 emphasised the need for tools that automate such procedures: *"it would be great to have a tool that stress tests the tests to ensure their stability"*. Indeed, the manual test validation could be assisted with variability-aware reruns that account for different configurations, inputs, and system states (*e.g.* [136]). These variations can build on the known causes of non-determinism (*e.g.* random inputs and the system resources) to expose, detect, and reproduce flaky tests.

Discussion: Our results confirm previous observations [49], [85], [137] and show that reproducing and debugging flaky tests remain the most challenging tasks for developers. Furthermore, our analysis accentuates the need for techniques and tools that monitor and analyse the system states to assist the prediction, debugging, and evaluation of flaky tests. This need is particularly relevant if we consider the results of RQ1, which suggested that flakiness can stem from the system interactions and factors that are external to the source code. Indeed, trace analysis could be a powerful tool that complements the current detection and prediction approaches, which rely mainly on the source code [59], [83], [91]–[93]. Our results also show that a more fine-grained analysis of flaky tests, using the flake rate, can be more insightful for developers. This aligns with the works that suggested that every test is potentially flaky [21], and research studies should focus on (or at least consider) the level of flakiness instead of classifying tests as flaky and non-flaky. Finally, our participants expressed the need for automating the quality assessment of software tests through static analysis and variability-aware reruns. In particular, techniques that rerun tests with different configurations or inputs, like Shaker [84] and FLASH [107], seem very promising if we consider the role of external factors on flakiness.

4.5 Threats to Validity

4.5.1 Transferability

A possible threat to the generalisability of our study is the number of participants. Unfortunately, due to the specificity of the topic, it was challenging to find developers qualified to take part in the study. We tried to ensure the quality of our results by only considering practitioners with relevant experience (with flakiness in particular and testing in general). The experience of our participants ranges from 6 to 35 years, with an average of 16 years. Our participants also constitute a diverse set of roles, company sizes, and application domains. Moreover, the collected data are enough to answer our research questions and provide us a theoretical saturation [116].

4.5.2 Credibility

A potential threat to the credibility of our findings could be the credibility of the analysed materials as we relied on grey literature and interview transcripts. In grey literature, we followed the quality assessment guidelines of Garousi *et al.* [109], which were specifically designed for such purposes. In interviews, we communicated the study objectives to the participants and clearly explained that the process is not judgemental. Moreover, we formulated our questions to target the practitioner experiences and observations.

4.5.3 Confirmability

A potential threat to the confirmability of our results is the accuracy of the analysis of the transcripts. To mitigate this threat, two authors performed consensual coding and all the authors discussed the coding guide iteratively, to ensure the clarity and precision of the identified sub-categories.

4.6 Conclusion

Our study shows that the analysis of flaky tests must consider the whole testing ecosystem and it should not be limited to the test and code under test. We also highlight a broader impact of flakiness on the testing practices and the overall system quality than what had been presented by previous work. Finally, we synthesise 16 measures adopted by practitioners to mitigate flakiness and we identify automation opportunities within them. These results open an avenue for future work:

- Flakiness stems mainly from the interactions between system components, the testing infrastructure, and uncontrollable external factors. Future studies can leverage monitoring and log analysis to propose techniques that assist practitioners in addressing flakiness.
- Establishing testing guidelines, *e.g.* recommendations on test size, external resources, and assertion thresholds, is a key measure for preventing flaky tests. Future studies can decrease the manual effort expended in enforcing such guidelines by providing static analysis tools and code review processes.
- Future work can leverage variability-aware reruns [136] and fuzzy testing to effectively expose and reproduce flaky tests. Such techniques can help in automating the current manual test validations performed by practitioners.
- Given the frequency of flaky tests and the cost of their mitigation, practitioners rely on the flake rate to adapt their strategies. Future work should account for this when assessing flaky tests and leverage it in their automated solutions.
- Some practitioners may falsely label buggy and non-deterministic features as flaky tests, and thus ignore them and treat them as false alerts. Future studies should further investigate the impacts of such confusions.

- Due to the difficulty of reproducing and debugging flaky tests, the fixing step is rarely achieved by practitioners. Future work should focus on providing tools that assist the root cause identification and reproduction of flaky tests.

5 A Replication Study on the Usage of Code Vocabulary to Predict Flaky Tests

In this chapter, we perform a replication of the vocabulary-based approach, one of the main techniques used to detect flaky tests. With this study, we intend to bring three contributions. First, we evaluate the approach under more realistic settings. Second, we check the generalisability of the approach by checking its application to another programming language. Finally, we experiment using an extended set of features in an attempt to improve the original approach.

This chapter is based on the work published in the following paper:

- G. Haben, S. Habchi, M. Papadakis, *et al.*, "A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests," *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021

Contents

5.1 Introduction

Regression testing is an important step of software development that ensures the stability of existing software features and allow multiple developers to work on a shared codebase. In typical large-scale development workflows, test suites are run after code changes to highlight any misbehaviour and validate new software releases. Unfortunately, software tests do not always give consistent results. This inconsistent behaviour is often referred to as test flakiness. Flaky tests exhibit a non-deterministic nature, *i.e.* they pass and fail for the same version of a program and the test [58].

Flakiness plagues Continuous Integration (CI) as tests are generally expected to pass in order to merge code changes [138]. Thus, flakiness introduces uncertainty, meaning that testers cannot be sure whether failures are true or not. Besides, flakiness affects productivity as developers invest time in reproducing and debugging flaky failures. Developers may also lose trust in their test suite and stop relying on it if there are too many false signals. Consequently, they could ignore failing tests that are caused by real defects in the program.

Several studies and reports from industrial actors have highlighted the prevalence and impact of flakiness [57], [85], [103].

A common approach to deal with flakiness is to rerun a failing test several times, hoping to expose non-deterministic behaviour. Unfortunately, these reruns imply cost both computationally- and time-wise. In the case of Google, this leads the company to spend between 2 and 16% of its computer resources rerunning flaky tests [20]. Many other companies report having problems dealing with flaky tests, including Huawei [52], Mozilla [53], Facebook [21] and Spotify [54].

To mitigate this problem, several strategies have been developed to detect flakiness. These can be divided into two main categories; the *dynamic approaches* that involve running the tests and analysing their outputs and logs over time, and the *static approaches* that attempt to identify flaky tests without any test execution.

Micco and Memon [56] presented a dynamic approach that identifies flaky tests by looking for specific patterns in the test execution outcomes observed in the recent development history (Pass to Fail, Fail to Pass), thereby proposing a simple pattern matching approach that achieves a 90% accuracy in classifying tests [56]. A similar approach was also presented by Apple [103], but the reality is that rerunning tests is still the main dynamic approach used to detect flakiness [55].

Pinto *et al.* [92] developed a prediction modelling approach that statically identifies flaky tests by analysing their code (test code only). This approach is appealing compared to the current practice due to its static nature that a) does not require any test execution logging and analysis that is usually hard to implement on the fly and usually not supported by the test infrastructures, and b) the low overhead it entails, i.e., it reduces the execution cost caused by test reruns.

The original study (Pinto *et al.*) evaluated the performance of different machine learning models on different representations of the test code and found that the vocabulary of tests can predict test flakiness with 95% of accuracy (F1 score). Although encouraging, the original study was performed in a dataset covering one programming language (Java), evaluated in a time-insensitive manner and left open many additional questions related to the vocabulary of source code. Considering the importance of the problem we decided to perform further investigations. We believe that extensive and independent evaluations are also necessary to reach industrial adoption and practice.

Replication is essential to verify experimental results from previous studies. They are a key aspect of empirical software engineering as they bring evidence that observations made can hold (or not) under other conditions. Different types of replication exist [139], [140]. An *exact* replication attempts to reproduce the experiments following as closely as possible the initial procedures. By doing so, we learn that the first results were not caused by uncontrolled random factors. In *conceptual* replication, one or more dimensions can be changed to investigate to what extent the results hold.

In this paper, we present a conceptual replication of the study of Pinto *et al.* [92]. We start by considering a different validation methodology than the one used in the study of Pinto *et al.* [92]. We thus adopt a time-sensitive validation setting that better reflects the envisioned use case of the approach; at a given point in time, we train our predictor with historically identified flaky tests and inspect the model performance in predicting unseen flaky tests, i.e., with the subsequent ”future” tests. We argue that this procedure is important to confirm the results and avoid biasing the predictions by considering future data.

Another aspect our study aims to evaluate is the generalisation of Pinto *et al.*'s findings, in particular to a different programming language. Therefore, we mine Python projects from GitHub and build a new dataset of 837 flaky tests. Then, we use it in order to evaluate the vocabulary-based flakiness prediction. This part of the analysis aims at re-validating the Pinto *et al.* findings on new and different data.

Finally, we go beyond the original study by considering an extended set of features. In particular, we attempt to predict flakiness using not only the vocabulary of test code (like Pinto *et al.* did) but also the Code Under Test (CUT). This endeavour follows the findings of many reports[58], [75], [141] revealing that much flakiness manifests in the CUT. Hence, we conduct a comparative study that highlights the impact of the two feature sets, *i.e.* , sources of vocabulary on flakiness prediction.

All in all, our results demonstrate that a more robust, time-sensitive validation has a consistent negative impact on the reported results of the original study (performance degrades by 7% on average) but, fortunately, do not invalidate the key conclusions of the study, *i.e.* , predictions are significantly better than random selections.

Additionally, we find re-assuring results that vocabulary-based models are more successful in Python than in Java (average performance of 80% in Python in contrast to 61% in Java), and perhaps surprisingly, that the information lying in the Code Under Test has a limited or no impact on the model performance. Taken together, these results corroborate the conclusion that the vocabulary of tests is indeed a viable and robust solution to the test flakiness problem.

5.2 The Original Study

This work is a replication of the study by Pinto *et al.* [92]. In this section, we briefly summarise the approach they presented for flakiness prediction. We first present the dataset of existing flaky tests which they used in their study. Then, we explain their source code representation and prediction model. Finally, we recall their evaluation methodology and results.

5.2.1 Dataset

In their original study, Pinto *et al.* relied on the DeFlaker dataset, which was proposed by Bell *et al.* [83]. This dataset includes 1,874 flaky tests identified using the DeFlaker tool on multiple revisions of 24 open-source Java projects. Pinto *et al.* selected 1,403 flaky tests from this dataset to build their set of flaky tests. They also randomly selected tests that were not flagged as flaky by DeFlaker to form a set of *a priori* non-flaky tests. To mitigate the problem of class imbalance, both sets had the same size.

5.2.2 Prediction Model

In order to prepare the classification inputs, Pinto *et al.* extracted identifiers that represent the test vocabulary and complexity. This extraction takes several steps. First, they localise the file where the test is defined. Then, they select all identifiers contained in this test, pre-process them by splitting them according to their camel-case syntax and converting them into lower-case. Finally, they remove stop words from the obtained set. Each flaky and non-flaky test is represented as follows:

- A vector of booleans indicating for each token if it is present in the test or not;
- The number of lines of code;
- The number of Java keywords contained in the test.

The last two features are used as a proxy for code complexity. The authors used

Table 5.1: Model performance of the Pinto *et al.* study [92]

Algorithm	Precision	Recall	F1	MCC	AUC
Random Forest	**0.99**	0.91	**0.95**	**0.90**	**0.98**
Decision Tree	0.89	0.88	0.89	0.77	0.91
Naive Bayes	0.93	0.80	0.86	0.74	0.93
Support Vector	0.93	**0.92**	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

these vectors as inputs for their prediction models. In particular, they evaluated the performance of five machine learning classifiers: Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour.

5.2.3 Evaluation

Evaluation methodology

The authors follow a standard methodology to train and evaluate the five classifiers. That is, they split the whole set of test cases into a training set containing 80% of the tests and a validation ("test") set containing the remaining 20%.

They report the standard precision, recall and F1-score metrics. The precision shows the proportion of correctly classified flaky tests. The recall shows the proportion of flaky tests found out of all existing ones. They focus their analysis on the F1-score, which combines precision and recall to assess the model performance. Detailed results for their different models are listed in Table 5.1.

Results

Among the five trained models, the most promising one was Random Forest, having a performance as high as 0.95 for the F1-score. Altogether, the five models showed great performance on their dataset.

5.3 The Replication Study

Our key goal is to investigate whether the conclusions of Pinto *et al.* generalize to different flakiness scenarios, viz., (1) a time-sensitive prediction use case where flakiness information about past tests are used to predict flakiness in future (new) tests, (2) flakiness prediction in different programming languages, (3) the use of different sets of features involving both test code and code under test. Each of these scenarios gives rise to a research question that we answer in our study. In all scenarios, we use the model presented in the original study that gave the best performance. The model is based on a bag of words and a Random Forest of

100 trees i.e., the model which gave the best results in the original study. Our replication package containing code, models and datasets is available online[1].

5.3.1 Research Questions

We aim at answering the following research questions:

RQ1: *How well do vocabulary-based models identify flaky tests when using a time-sensitive validation?*

RQ2: *How well do vocabulary-based models identify flaky tests in other programming languages?*

RQ3: *Is the vocabulary of Code Under Test useful for flakiness prediction?*

5.3.2 RQ1: Time-Sensitive Validation

In the real world, one can picture different usages for a flaky test prediction model. For instance, in Continuous Integration (CI) environments where new changes (commits) making some tests fail are typically rejected, developers can ignore those failing tests that are likely to be flaky and isolate them for further investigation.

In another setting, the prediction model can also come as an IDE plugin hinting at tests that use keywords related to flakiness.

These scenarios illustrate the importance of the temporality of tests and code, as the model can be trained only on flaky tests detected previously to predict new occurrences. Moreover, the fact that the vocabulary of code changes as new commits are introduced makes it challenging for models trained on older data to predict flakiness in future code versions that are temporarily distant.

The model can also be limited to flaky tests detected in one project, e.g., when the vocabulary linked to flakiness can differ from one project to another. Indeed, as reported in the literature [58], [85], different sources of flakiness exist such as concurrency issues, usage of date/time, I/O actions, API or network calls, etc. Thus, based on the project, the flakiness sources can differ and the vocabulary associated with it varies accordingly.

For all these reasons, we propose a novel, intra-project, time-sensitive setup for validating flakiness prediction models. This setup evaluates a model on its ability to predict new flaky tests with data that is assumed to be known from the past of the project.

To compare this setup with the one from Pinto *et al.*, we rely on the DeFlaker dataset, which was also used in the original study. For each project, we select tests that were found flaky at any revision of the change history to form the Flaky Tests set FT. DeFlaker does not provide explicit information about tests that did not flake, as the tool can not guarantee that a test that did not fail (yet) is not

[1]`https://github.com/serval-uni-lu/FlakyVocabularyReplication`

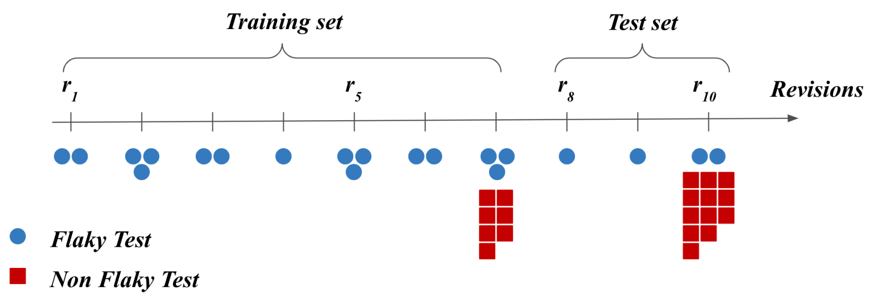


Figure 5.1: Time-sensitive validation

flaky. We define Non Flaky Tests NFT as tests that were not found as flaky in any revision, that is, $NFT = T_{Total} - FT$.

Figure 5.1 explains how FT and NFT are selected in our time-sensitive validation. We split our dataset in order to have 80% of the FT from earlier revisions for our training set and 20% of the FT from "future" revisions for our test set.

We select the NFT from the revision where the last FT_{train} are selected for the training set and from the last revision where FT_{test} are selected for the test set.

To assess the impact of this new setup on model performance, we compare it with a classical setup where the model is trained and tested with flaky tests regardless of their observation date (i.e., the setup followed by Pinto *et al.*). In such setup, all flaky and non-flaky tests are grouped without accounting for their observation date. Then, the groups are randomly split into training and test sets following an 80/20 ratio.

To perform this comparison, we selected six projects from the DeFlaker dataset based on their numbers of flaky tests. These projects have at least 30 flaky tests, which we consider as a minimum necessary for training and testing a model. Table 5.2 presents these projects with their numbers of flaky and non-flaky tests. We also present the dates of the first and last flaky tests identified in these projects. We split this dataset according to the two validation setups, then we build our prediction model, train it and contrast the results of both setups.

5.3.3 RQ2: Generalisation to other Programming Languages

Predicting flaky tests in Python

Another goal of our study is to evaluate the generalisability of the original study to other programming languages. For this purpose, we propose to assess the performance of flakiness prediction models on Python projects. We chose Python because it is the most popular language used in modern projects and it is commonly used for machine learning, web development, game development, and many other

Table 5.2: Details about the Java projects used in our study

Project	**Earliest revision**	**Latest revision**	**#FT**	**#NFT**
achilles	2015-10-30	2016-09-05	51	392
hbase	2010-05-17	2010-06-21	98	120
okhttp	2014-03-06	2015-01-30	102	1178
oozie	2013-03-20	2013-05-31	1039	44
oryx	2015-01-06	2015-02-27	38	286
togglz	2016-01-23	2016-06-17	20	256

applications.

Python comes with its set of testing frameworks. We focus our study on Pytest[142]. Pytest is the equivalent of Junit for Python and enables developers to write tests for their programs. It is one of the main testing frameworks used in the open-source community and in the industry. Pytest comes with its lot of features and plugins. Especially, a specific module to handle flaky tests can be used with Pytest: flaky[2]. This module allows developers to annotate tests as flaky to automatically rerun them in case of failure. The developer can also configure the maximum amount of reruns to attempt and the minimum number of passes required. This annotation can be added to a test function or directly to the test class, giving its property to all of its tests. Figure 5.2 shows an example of a test marked as *@flaky* taken from the Typed_python project[3].

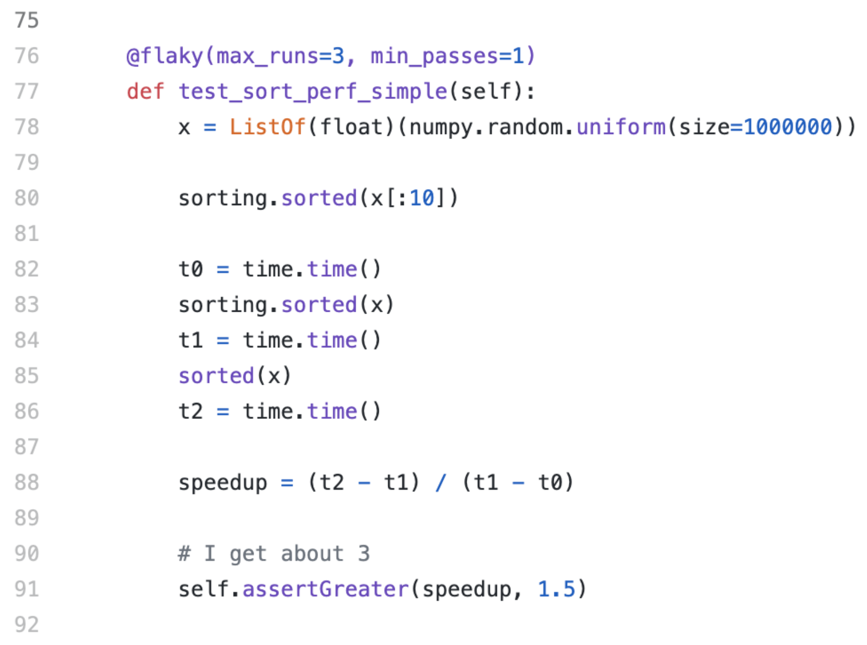
We mined GitHub using the source-graph API[4], searching for Python projects containing the annotation *@flaky.* This process yielded 110 projects with a total of 1,304 tests marked as flaky. Similarly to our first experimentation, we only select projects in which we have enough flaky tests to train and test a model, *i.e.* 30 flaky tests. This results in a dataset of 9 projects and 837 tests marked by developers as flaky. Table 5.3 shows these projects with their number of flaky and non-flaky tests. Compared to the Java dataset, we were able to obtain more projects with more flaky tests for our study. To the best of our knowledge, this is the first dataset of flaky tests in Python.

It is worth noting that in this research question, we evaluate the performance of the model to predict flaky tests in a single revision. Therefore, we reuse the typical 80/20 dataset split as followed by Pinto *et al.*. That is, we are rather focusing on confirming that the approach works as well in Python and that a model can learn features differentiating tests labelled as *@flaky* from the ones that are not. To

[2]`https://pypi.org/project/flaky/`

[3]`https://github.com/APrioriInvestments/typed_python`

[4]`https://sourcegraph.com/search`



```python
@flaky(max_runs=3, min_passes=1)
def test_sort_perf_simple(self):
    x = ListOf(float)(numpy.random.uniform(size=1000000))

    sorting.sorted(x[:10])

    t0 = time.time()
    sorting.sorted(x)
    t1 = time.time()
    sorted(x)
    t2 = time.time()

    speedup = (t2 - t1) / (t1 - t0)

    # I get about 3
    self.assertGreater(speedup, 1.5)
```

Figure 5.2: Example of a test labelled @flaky

extract these features, we use a bag of words representation of the test, as in Java. We also carefully remove the *@flaky* annotations, as keeping it in the vocabulary would bias our model towards recognising this annotation rather than the code vocabulary.

Predicting manifest flaky tests

We perform further analysis in Python to assess the usefulness of a vocabulary-based model. Our objective is to evaluate the ability of a model to identify *manifest* flaky tests based on training with tests labelled as flaky by developers. We consider as manifest flaky, every test for which we are able to observe non-deterministic behaviour dynamically. This means that the test fails and passes at least once after several reruns. To identify these manifest flaky tests, we reran 200 to 300 times the test suite of the three projects Bokeh, Celery and Python-telegram-bot. We run the test suites on a Mac machine with a 2,4 GHz 8-Core i9 processor and 32Gb of RAM. The results of these reruns are presented in the table 5.4. The column #@flaky shows the number of tests labelled as flaky in each project.

We observe that despite the high number of reruns (800), only 23 tests have a flaky behaviour. This outcome is not surprising as flaky tests are, by nature, difficult to reproduce. To assess the model performance in detecting manifest flaky tests, we focus on the only project that has a reasonable amount of manifest flaky tests, namely Python-telegram-bot. We use the 20 manifest flaky tests found

Table 5.3: Python projects used in our study

Project	SHA	#FT	#NFT
bokeh	ddc22b8	100	2505
cassandra-dtest	8cb6bd2	72	4221
celery	0833a27	54	2890
jira	7fa3a45	131	59
pipenv	8e64873	32	1612
python-amazon	84c16f5	35	15
python-telegram-bot	8e7c0d6	186	1382
spyder	413c994	173	1086
typed-python	96e7ebd	54	6034

Table 5.4: Classifier performance for Python projects with manifest flaky tests

Project	#reruns	#@flaky	#manifest FT
bokeh	200	100	1
celery	300	54	2
python-telegram-bot	300	186	20

during our reruns as a test set, completed by 20 randomly selected tests that are not labelled as flaky. For the training set, we use the flaky and non-flaky tests minus the tests present in the test set.

5.3.4 RQ3: Extended Set of Features

So far, the flakiness prediction is only based on features taken from the test code. However, flaky tests can be due to infrastructure or environmental issues (*e.g.* lack of available resources in the CI, service or network unavailable, etc), to the test itself (*e.g.* usage of dates, randomness, order dependency, etc), or to the CUT (*e.g.* non-determinism, concurrency, etc). Notably, Luo *et al.* [58] showed that 24% of the fixes for flaky tests were applied to the CUT and that among them, 94% fixed a bug in the CUT. Hence, it can be judicious to consider information from the CUT in flakiness prediction models. We propose to extend the original study by including the vocabulary of the CUT in test representation.

The main issue when considering the CUT is that computing the code coverage of each test during each revision would bring significant overheads. Besides, retrieving the exact code coverage dynamically goes against the goal of static prediction, which is to reduce dynamic costs. To avoid this overhead, we propose a lightweight approach that relies on Information Retrieval (IR) to estimate the CUT.

IR techniques have been used to solve different software engineering problems [143]–[145]. IR aims at quickly and automatically retrieving relevant information among a set of documents based on keywords taken from a user query. In our case, the query is the tokens of a test and the set of documents is the set of all functions (or methods) defined in the project. Our hypothesis is that functions from the CUT of a test are likely to use similar keywords (*i.e.* variable names, API calls, etc) as the test. We are then looking for the most similar functions to our test function. To do so, we use a cosine similarity between a test case and a function from the CUT. Cosine similarity is defined with:

$$cosSimilarity = \cos(Tc, Func) = \frac{Tc \cdot Func}{|Tc||Func|}$$

where Tc is the vector representing the test code and $Func$ is the vector representing the function code. The result of a cosine similarity ranges from -1, meaning that the query - our test case - is completely different from the document - our function - to 1, where the query is perfectly similar to the document. In our case, we select the top three most similar functions for each test.

Algorithm 1 describes the process of associating the CUT to each test. In order to compute the cosine similarity between the test and a function, we use the Text tokenisation utility class from the Keras library[5]. We first fit the *Tokeniser* with the vocabulary from all tests and functions. Then, we transform the text of test and function bodies by creating a vector for each one of them of a length equal to the size of the vocabulary. In this vector, each element represents the number of times a word appears in the body. After extracting the vectors, we compute the cosine similarity between the current test and all functions and store results. We finally filter to only keep functions that have a high score, *i.e.* that they are the closest to the test. The body representation of the selected functions is used as a new set of features for the flakiness prediction model.

5.4 Results

5.4.1 RQ1: Time-Sensitive Validation

In this research question, we compare prediction model performance using a time-sensitive validation and a classical validation.

Figures 5.3-5.5 show the performance of our Random Forest classifier under time-sensitive and classical validation. Overall, we observe that the validation setup has an impact on the classifier performance. This impact varies significantly depending on the project, its size, and history of flaky tests. The projects Achilles, Hbase, OkHttp, and Togglz observe a decrease in their MCC score. The largest

[5]`https://keras.io/api/preprocessing/text/`

Algorithm 1: Cost effective retrieval of the CUT

```
Inputs:
Test[]
Function[]
Outputs:
TestWithCUT[]
Procedure CUT_SELECTION(Test[], Function[])
foreach test $T \in$ Test[] do
    similarityMeasures[]
    Tvector = transform(T)
    foreach function $F \in$ Function[] do
        fit($T + F$)
        Fvector = transform(F)
        cosTF = cosSimilarity(Tvector, Fvector)
        similarityMeasures.Append(cosTF)
    end
    similarityMeasures.Sort()
    similarityMeasures.Slice(0, 2)
    T.append(similarityMeasures)
    TestWithCUT.append(T)
end
return TestWithCUT[]
```

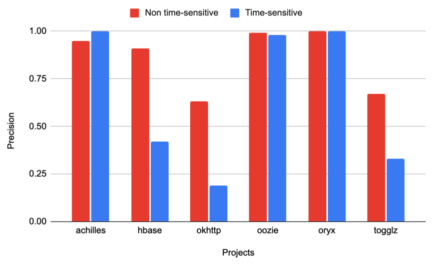


Figure 5.3: Precision under classical and time-sensitive validations.

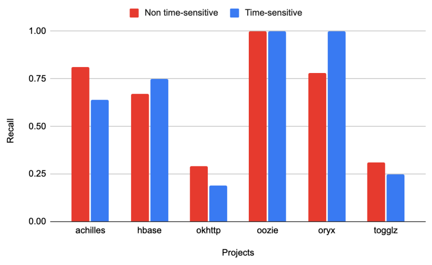


Figure 5.4: Recall under classical and time-sensitive validations.

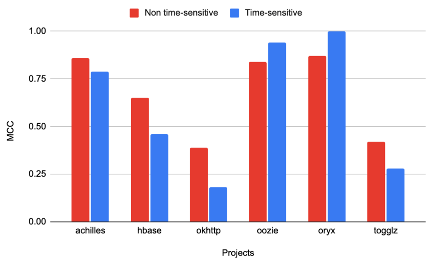


Figure 5.5: MCC under classical and time-sensitive validations.

performance drop is observed in the OkHttp project, where the MCC dropped from 0.39 to 0.18. The two exceptions are for Oozie and Oryx, where MCC increased by 0.10 and 0.13 points respectively. In the case of Oryx, this can be explained by the fact that most of the flaky tests come from one revision, thus, the time-sensitive validation has little to no impact. The difference can then be explained by the random selection of the samples when splitting the training and test set. The phenomenon is only present for this project. In the case of Oozie, there is a considerable imbalance between the number of flaky tests (1039) and non-flaky tests (44). Hence, the test set contains only 9 non-flaky tests, which might not be enough to draw conclusions.

RQ1: The performances of a vocabulary-based model decrease under a time-sensitive validation (MCC value drops up to 0.21). Nonetheless, the approach is still able to decently predict flaky tests.

Table 5.5: Classifier performance for Python projects

Project	Precision	Recall	F1	MCC	AUC
bokeh	**1.00**	**0.91**	**0.95**	**0.95**	**0.95**
cassandra-dtest	**0.96**	0.43	0.58	0.63	0.71
celery	0.85	0.54	0.64	0.66	0.77
jira	**0.98**	**0.99**	**0.99**	**0.95**	**0.98**
pipenv	0.78	0.19	0.30	0.37	0.60
python-amazon	**0.97**	**1.00**	**0.99**	**0.95**	**0.96**
python-telegram-bot	**1.00**	**0.99**	**1.00**	**0.99**	**1.00**
spyder	**0.92**	0.77	0.83	0.82	0.88
typed-python	**1.00**	0.86	**0.91**	**0.92**	**0.93**

5.4.2 RQ2: Generalisation to other Programming Languages

Predicting flaky tests in Python

Table 5.5 reports on the model performance when predicting flaky tests in 9 Python projects.

First, we observe that for 5 projects out of 9, the model reaches a great performance with MCC values greater than 0.9. For the rest of the projects, these scores are always higher than 0.50, except for Pipenv, which shows the lowest results with an MCC value of 0.37. Similarly, all the studied projects have a F1-score greater than 90% and 7 out of the 9 studied projects have a precision higher than 60%. These observations show that the vocabulary-based model is able to predict flaky tests with decent performance in Python projects.

Predicting manifest flaky tests

Table 5.6 shows the model performance in detecting manifest flaky tests based on tests marked as flaky by developers. The results show a perfect performance with MCC and F1-score values of 1 and 100% respectively, confirming that a model trained on tests labelled by developers can be used to predict manifest flaky tests. Interestingly, 2 of the 20 manifest tests were not labelled as flaky by the developers and were only identified with the reruns. Yet, the model was able to predict them by only learning from tests marked by developers. In a real-world scenario, we could picture the model finding those tests and automatically annotating them.

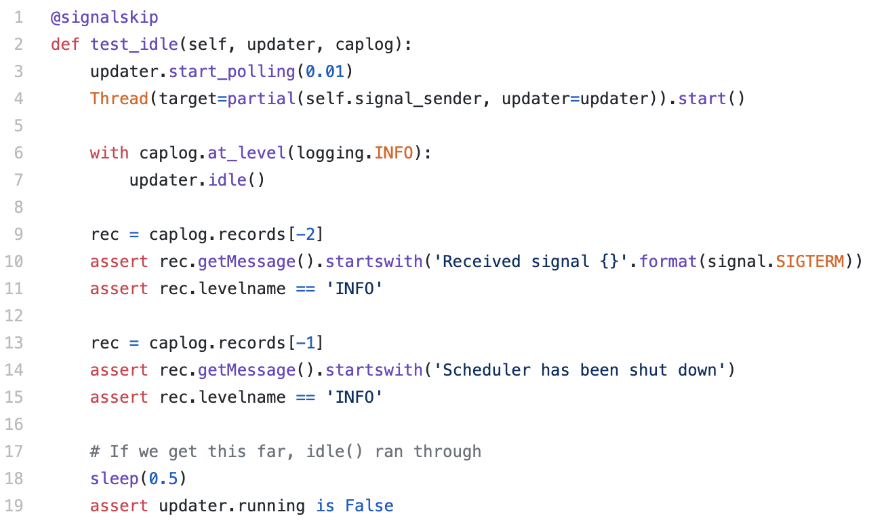
Figure 5.6 shows the test `test_idle()` from the class `TestUpdater`[6]. Over 300 reruns, this test failed intermittently because of a concurrency issue where a

[6]https://github.com/python-telegram-bot/python-telegram-bot

Table 5.6: Classifier performance for manifest flaky test in the Python-telegram-bot project

Project	**Precision**	**Recall**	**F1**	**MCC**	**AUC**
python-telegram-bot	1.00	1.00	1.00	1.00	1.00

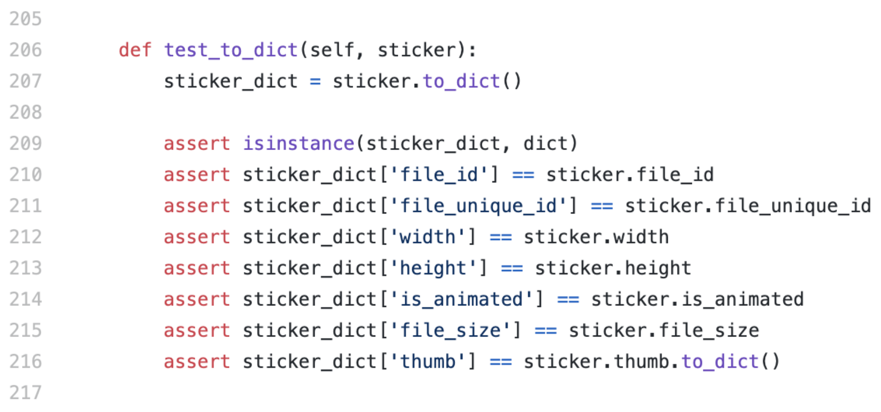
scheduler has been shut down. Indeed, the test body contains several keywords related to time and concurrency, which are common causes of flakiness, *e.g.* `Thread, sleep, idle`. In order to understand how the model predicted that this test is flaky, we analyse the most important features of the model. These features do not completely reflect the model prediction and they can be biased [146], but they give us an idea of the vocabulary that the classifier is using for its predictions. In the project Python-telegram-bot, we found that the top ten features include the keywords: `process, timeout, duration, seconds`, which are also related to time and concurrency. Hence, the model's ability to predict the test flakiness based on the vocabulary.



```python
@signalskip
def test_idle(self, updater, caplog):
    updater.start_polling(0.01)
    Thread(target=partial(self.signal_sender, updater=updater)).start()

    with caplog.at_level(logging.INFO):
        updater.idle()

    rec = caplog.records[-2]
    assert rec.getMessage().startswith('Received signal {}'.format(signal.SIGTERM))
    assert rec.levelname == 'INFO'

    rec = caplog.records[-1]
    assert rec.getMessage().startswith('Scheduler has been shut down')
    assert rec.levelname == 'INFO'

    # If we get this far, idle() ran through
    sleep(0.5)
    assert updater.running is False
```

Figure 5.6: A manifest flaky test not labelled @flaky

Figure 5.7 shows the test `test_to_dict()` from the class `TestStickerSet`, which is also manifestly flaky but the developers did not mark it as such. Unordered collections have been identified as a cause of flakiness by several works as developers can wrongly assume that elements of a collection will be returned in a specific order [58], [107]. In Python, the return order of dictionaries has varied over the different versions [147], [148]. In our case, we found that the keyword `dict`, which is present in a large number in this test, was among the first eight most important features of our classifier. This feature allowed the vocabulary-based model to predict that this test is flaky.



```

    def test_to_dict(self, sticker):
        sticker_dict = sticker.to_dict()

        assert isinstance(sticker_dict, dict)
        assert sticker_dict['file_id'] == sticker.file_id
        assert sticker_dict['file_unique_id'] == sticker.file_unique_id
        assert sticker_dict['width'] == sticker.width
        assert sticker_dict['height'] == sticker.height
        assert sticker_dict['is_animated'] == sticker.is_animated
        assert sticker_dict['file_size'] == sticker.file_size
        assert sticker_dict['thumb'] == sticker.thumb.to_dict()

```

Figure 5.7: A manifest flaky test not labelled @flaky

We conclude that the approach is extendable to the Python language, supporting the idea that the vocabulary-based prediction can be generalisable to other projects and programming languages. Moreover, we saw that we can take advantage of a flaky tests classifier using vocabulary-based features in order to identify vocabulary linked to flakiness and help developers write better quality tests.

RQ2: Vocabulary-based models can be generalised to other projects and programming languages. Besides, these models can leverage annotated flaky tests to predict and annotate manifest flaky tests that were not known to developers.

5.4.3 RQ3: Extended Set of Features

Figures 5.8-5.10 show the results of our prediction model in Java projects, while figures 5.11-5.13 present the model performance in Python projects.

In figures 5.8-5.10, we observe that the impact of including the CUT does not have a consistent impact on the model performance in Java projects. Adding the CUT improves the model performance in Hbase, Okhttp and Togglz, with an increase of the MCC value between 0.01 and 0.07. However, the opposite effect is observed in the projects Achilles and Oryx where the MCC dropped by 0.06 and 0.02 respectively. As for the Oozie project, including the CUT does not seem to impact the model performance. Nonetheless, these performance improvements and losses remain minor in all the studied Java projects.

Figures 5.11-5.13 show a similar effect of the CUT usage in Python projects. Out of the nine studied, six projects report a lower performance when adding the CUT to the features. This performance loss is up to 0.13 (MCC) in the projects

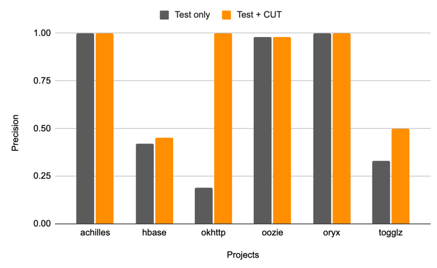


Figure 5.8: Precision score in Java projects

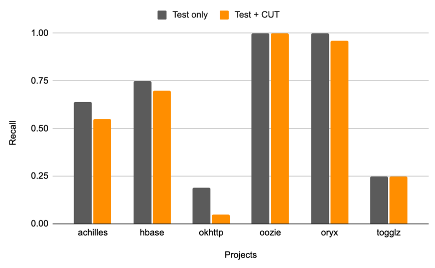


Figure 5.9: Recall score in Java projects

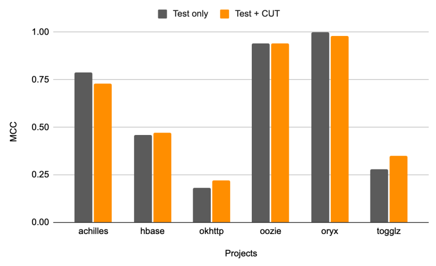


Figure 5.10: MCC score in Java projects

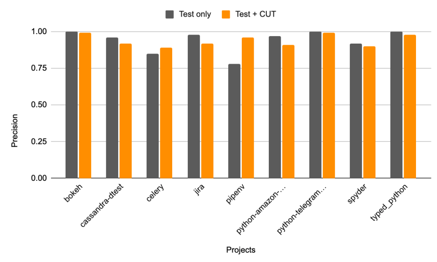


Figure 5.11: Precision score in Python projects

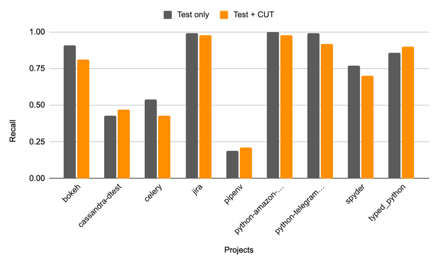


Figure 5.12: Recall score in Python projects

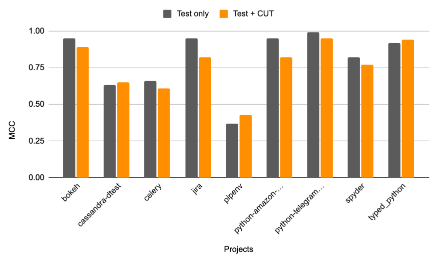


Figure 5.13: MCC score in Python projects

Jira and Python-amazon. On the other hand, the projects Cassandra-dtest, Pipenv, and Typed_python have better predictions when the CUT is used — an increase in the MCC value by 0.02, 0.06 and 0.02 respectively.

Based on the observations in both Java and Python, we conclude that including the CUT does not consistently improve the performance of a vocabulary-based model for predicting flaky tests.

RQ3: Surprisingly, the vocabulary of the Code Under Test, which is commonly considered as a source of flakiness, does not improve the performance of flakiness prediction models.

5.5 Threats to Validity

5.5.1 Construct Validity

One possible threat to the study's construct validity is our choice and selection of flaky tests in Python. It is possible that tests that are marked as flaky by developers are not actually flaky. In particular, developers could abuse of the annotation and mark non-flaky tests to forecast flaky behaviour. To inspect this point, we manually analysed projects from our dataset to check if this behaviour is prevalent. We found that some projects (like Jira and Python-telegram-bot) use the annotations to mark all class tests as flaky. However, this usage seems judicious as the class tests performed GUI testing, which is known for being a major cause of flakiness. Moreover, an abusive usage of this annotation by developers seems unlikely considering the rerun costs. When running the test suites, we observed that one pass can take a long time. Hence, it is not in the best interest of developers to mark as many tests as flaky to anticipate flakiness as this would largely increase the execution time as soon as there are test failures. We believe that the usage of annotated flaky tests in our study is reasonable given the lack of large datasets of flaky tests, especially for programming languages other than Java. Ideally, the annotated flaky tests would be validated by rerunning them and exhibiting their non-deterministic behaviour. Nevertheless, the reproduction remains very challenging for flaky tests in general and even tests identified in other datasets are hardly reproducible [92], [137].

Another threat to construct validity could be the approach we use to retrieve the CUT. Intending to design a fast and lightweight approach, we used Information Retrieval to estimate the real code coverage of each test. This approximation can be responsible for the noise brought in the features. To investigate this point, we assessed the CUT effect when using other retrieval approaches. First, we retrieved an approximation of the CUT by using Static Call Graph. We selected all functions called by the test as the CUT and we do not explore what those functions call.

Even if this approach only includes a subset of the CUT and flakiness can be caused by functions deeper in the call graph, we believe that keywords in the top functions should serve as a proxy. Results for this approach were similar to the ones presented in RQ3. The performance scores slightly decrease or increase from one project to another without showing a significant impact on the prediction performance. Furthermore, we computed the code coverage for projects where we managed to build and run the test suite. This task is challenging, especially in Java where the flaky revisions are from several years ago and dependencies are easily missing from central repositories. We successfully retrieved the real code coverage for revisions of Togglz and Oryx using the GZoltar tool [149]. This tool allows us to get a coverage matrix representing each line covered by the test case. We used this matrix to retrieve the exact CUT and include it as a feature for our prediction model. For both projects, the CUT inclusion had an impact on the model performance, which is very similar to the one observed with the CUT retrieved with IR. Hence, we believe that the results observed in RQ3 are not flawed by the CUT retrieval.

5.5.2 Internal Validity

One possible threat to internal validity is the definition of non-flaky tests. The datasets that we used for both Java and Python, only mark flaky tests and do not provide information about non-flaky tests. Consequently, we considered all tests that were not marked as flaky to be non-flaky. Yet, some of these tests can be flaky even though DeFlaker or the developer did not mark them as such. This limitation is not unique to our study as it is theoretically impossible to prove that a test is not flaky. To the best of our knowledge, there are no datasets, neither in formal nor in grey literature, that mark explicitly non-flaky tests. On top of that, our study results show that there is a clear distinction between the classes of flaky and non-flaky tests. Accordingly, it is unlikely that a significant fraction of the non-flaky tests is actually flaky.

One common threat to the internal validity of replication studies is potential errors in the reproduction (*e.g.* settings and library usage). To alleviate this threat, we carefully examined the GitHub repository of the original work [150] to understand and reproduce their implementation details. Besides, the goal of our study is not to exactly reproduce the original work and our results align well with the original findings.

5.5.3 External Validity

The main threat to our external validity is the size and nature of our datasets. For Java, we relied on the DeFlaker dataset since it is the largest open-source set of flaky tests and it has already been used in many flakiness studies [92], [93]. As for Python, we built a dataset of 837 flaky tests from 9 projects by mining GitHub repositories. For the sake of generalisability, it would have been preferable to include more projects and flaky tests. Nevertheless, our intra-project setting required a minimum number of flaky tests per project and limited our choices. We encourage future studies to replicate this study on larger datasets, including industrial projects.

5.6 Conclusion

This paper explored the usability and performance of vocabulary-based models in predicting flaky tests. We presented a conceptual replication of the study of Pinto *et al.*, following three axes.

- First, we evaluated the prediction performances under a time-sensitive validation setting that better reflects the envisioned use case for the approach. We found that a more robust validation has a consistent negative impact on the reported results of the original study (performance degrades by 7% on average). Fortunately, this performance degradation does not invalidate the key conclusions of the study as the model predictions are significantly better than random selections.
- Second, we evaluated the generalisability of a vocabulary-based model to other programming languages. We found re-assuring results that vocabulary-based models are more successful in Python than in Java (average performance of 80% in Python in contrast to 61% in Java). We also showed that these models can leverage flaky tests annotated by developers to predict and annotate manifest flaky tests that were not known to developers.
- Third, we conducted a comparative study that highlights the impact of features lying in the CUT on the prediction performance. Surprisingly, we found that the vocabulary of the CUT, which is commonly considered as a source of flakiness, does not improve the performance of vocabulary-based models.

On top of these findings, this paper presents a new large dataset of flaky tests mined from developer annotations in Python projects on GitHub. This dataset and our experiment toolset are available in a comprehensible replication package.

6 Predicting Flaky Tests Categories using Few-Shot Learning

The last two chapters explored the challenges linked with test flakiness and in particular flaky test prediction. We identified the need for both researchers and developers to better understand the root cause of a flaky test once detected. Thus, we present in this chapter FlakyCat, the first approach to classify flaky tests based on their category of flakiness. This technique, based on CodeBERT for source code representation, leverages few-shot learning and Siamese networks to learn from a limited set of examples. To enable a better comprehension of the predictions, we also introduce an interpretability technique for CodeBERT-based models.

This chapter is based on the work published in the following paper:

- A. Akli, G. Haben, S. Habchi, *et al.*, "Predicting flaky tests categories using few-shot learning," in *Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test*, ser. AST '23, Melbourne, Australia: Association for Computing Machinery, 2023, ISBN: 9781450392860

Contents

6.1 Introduction

Continuous Integration (CI) plays a key role in nowadays software development life cycle [10], [138]. CI ensures the quick application of changes to a main code base by automatically running a variety of tasks. Those changes are responsible for building the program and its dependencies, performing checks (*e.g.* static analysis), and running test suites to maintain code integrity and correctness. An important assumption for practitioners is that tasks are deterministic, *i.e.* regardless of the execution's context of a same task, results need to remain similar.

Unfortunately, in practice, this is not always the case. Previous research has identified test flakiness as one of the main issues in the application of automated software testing [20], [57], [151]. A flaky test is a test that passes and fails when executed on the same version of a program. Flakiness hinders CI cycles and prevents automatic builds due to false signals, resulting in undesirable delays. Furthermore, surveys [49], [81], [105] show that flakiness affects developers' productivity, as they spend a considerable time and effort investigating the nature and causes of flaky tests.

To alleviate this issue, researchers have proposed tools that help detect flaky tests. In particular, IDFlakies [59] and Shaker [84] detect flakiness in test suites by running tests in different setups. However, rerunning tests, especially for a large number of times, is resource-intensive and might not be a scalable solution. For this reason, researchers recently suggested alternative approaches to detect flaky tests based on features that do not require any test execution [57], [91], [92], [152]. Although promising, these approaches mainly focus on classifying tests as flaky or not without any additional explanation. Unfortunately, the absence of additional information prevents a proper comprehension of flaky failure causes. Hence, further investigation is required to understand the nature of flakiness and identify the culprit code elements that need to be fixed [49].

Another important line of research in the area regards automated approaches that aim at helping to locate the root causes and suggest potential flakiness fixes [85], [86], [89]. However, research on automatically fixing flakiness is still at an early stage: tools often focus on one category of flakiness and with few examples. For instance, iFixFlakies [87] and ODRepair [88] focus only on dealing with test order dependencies, which is one of the main causes of test flakiness. Flex [107] automatically fixes flakiness due to algorithmic randomness in machine learning algorithms.

We believe that both developers and researchers would benefit from additional information that could assist them in gaining a better understanding of flaky tests, once they have been detected. Therefore, we propose FlakyCat, a learning-based flakiness categorisation approach that identifies the key reason/category of the test failures.

One limitation of previous work, relying on supervised learning, regards the need for large volumes of available data. Unfortunately, debugged flaky test data is scarce, inhibiting the application of learning-based methods. To deal with this issue, we leverage the Few-Shot learning capabilities of Siamese networks, which we combine with the CodeBERT representations to learn flakiness categories from a limited set of data (flaky tests).

To evaluate FlakyCat, we gather a set of 451 flaky tests annotated with their category of flakiness issued from previous studies and projects that we mined from GitHub.

Our empirical evaluation aims at answering the following research questions:

RQ1: *How effective is FlakyCat compared to approaches based on other combinations of test representation and classifier?*
Findings: Our results show that FlakyCat is capable of predicting flakiness categories with an F1 score of 73%, outperforming classifiers based on traditional supervised machine learning.

RQ2: *How effective is FlakyCat at predicting each of the considered flakiness categories?*
Findings: FlakyCat classifies accurately flaky tests related to *Async waits*, *Test order dependency*, *Unordered collection*, and *Time*, with the best F1 score of 81% for the *Async waits* category. However, the approach shows difficulty in classifying concurrency-related flaky tests (an F1 score of 39%), since these cases are related to the interaction of threads and processes and are easily confused with Asynchronous waits.

RQ3: *How do statements of the test code influence the predictions of FlakyCat?*
Findings: We found that some statement types are specific to certain flakiness categories. This is the case for assert statements in *Unordered collections* and statements using date or time for the *Time* category. We also found that some flaky categories have similar statement types like the presence of thread usages in both *Async waits* and *Concurrency* categories.

In summary, our contributions can be summarized as follows:

Dataset We collected 451 flaky tests alongside their categories.
Model We present FlakyCat, a new approach using Few-Shot Learning and CodeBERT to classify flaky tests based on their flakiness category.
Interpretability We introduce a novel technique to explain what information is learnt by models using CodeBERT as code representation.

To enable the reproducibility of our work, we make the dataset used to evaluate

FlakyCat and the scripts publicly available in our replication package [1].

The paper is organized as followed: Section 6.2 presents the designed implementation of FlakyCat. Section 6.3 introduces our interpretability technique. Section 6.4 describes how we collected our dataset and evaluated our study. Section 6.5 presents the results of our study. We further discuss different use cases in Section 6.6. Finally, Section 6.8 discusses threats to the validity of this study.

6.2 FlakyCat

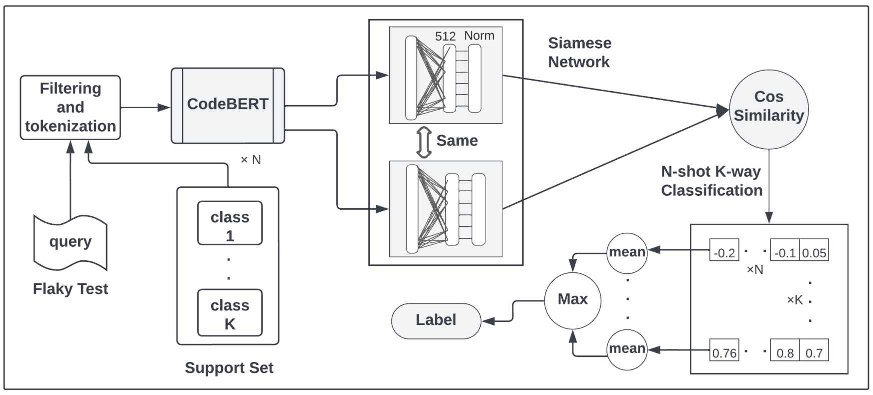


Figure 6.1: An overview of FlakyCat, which combines the use of the pre-trained model CodeBERT, and Few Shot Learning based on the Siamese network.

In this section, we present the design and implementation of our approach. Figure 6.1 presents an overview of the main steps of FlakyCat, code transformation and classification.

6.2.1 Step 1: Flaky Test Transformation

Scope

We rely on the test code to assign flaky tests to different categories. Previous studies showed that flakiness finds its root causes in the test in more than 70% of the cases[58], [62]. Hence, focusing on the test code allows us to capture the nature of flakiness while minimizing the overall cost of FlakyCat. Indeed, considering the code under test would require running the tests and collecting the coverage, which entails additional requirements and costs.

Flaky Test Vectorisation

In order to perform a source code classification task, we first need to transform the code into a suitable representation that will be fed to the classification model.

[1] https://github.com/serval-uni-lu/FlakyCat

Among previous studies predicting flaky tests statically, two main approaches were used to transform code into vectors: using test smells [96], [152] and using code vocabulary [92], [94], [95]. Both approaches seem promising, as different studies report high-performance models. As their encoding enables flaky test prediction, we believe they could also be used for flakiness category prediction, and we compare them with our approach.

Recently, code embeddings from pre-trained language models were also considered for source code representation [97], [153]. Pre-trained language models allow the encoding of code semantics and are intended for general-purpose tasks such as code completion, code search, and code summarisation. Considering these benefits, we use the pre-trained language model CodeBERT [154] to generate source code embeddings. CodeBERT can learn the syntax and semantics of the code and doesn't require any predefined features [155]. Considering this aspect, we decide to rely on the CodeBERT test representation.

CodeBERT has been developed with a multi-layer transformer architecture [156] and trained on over six million pieces of code involving six programming languages (Java, Python, JavaScript, PHP, Ruby, and Go).

To get the code representation using CodeBERT model, we first filter out extra spaces such as line breaks and tabs from the source code. In our case, we use each test method's source code as individual sequences. We then tokenise sequences by converting each token into IDs. Each sequence is passed to the CodeBERT model, which returns a vector representation. Figure 6.2 illustrates this process.

Next, we explain the inputs and outputs of CodeBERT.

Inputs CodeBERT is able to process both source code and natural language, *e.g.* comments and documentation. In our case, we did not exploit the possibility of using comments as the input length of CodeBERT is limited. Furthermore, comments can add noise since they represent unstructured text, possibly written by different developers, so we decided to solely rely on the code semantics. Hence, the given input to CodeBERT only considers code tokens, surrounded by two special tokens for boundaries. This is represented as follows:

$$[CLS], c1, c2, ..., cm, [SEP].$$

Where Ci is a sequence of code tokens, the special token [SEP] indicates the end of the sequence, and [CLS] is a special token placed in the beginning, whose final representation is considered as the representation of the whole sequence which we use for classification.

Outputs CodeBERT output includes two representations. The first one is the context matrix where each token is represented by a vector, and the second one is the CLS representation, having a size of 768, which is an aggregation of the context matrix and represents the whole sequence. For the purpose of FlakyCat, we are interested in the CLS vector that represents the complete test code.

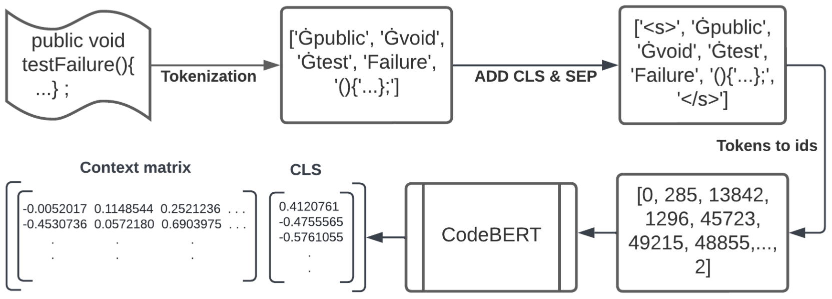


Figure 6.2: The process of converting the source code of each test case to a vector using CodeBERT, going through tokenisation, then converting to IDs and applying the CodeBERT model to get the representation (CLS vector). Ǵ represent spaces, $< s >$ used for CLS, and $< /s >$ for SEP.

6.2.2 Step 2: Flaky Test Categorisation

Classification process

Unlike traditional machine learning classifiers that attempt to learn how to match an input x to a probability y by training the model in a large training dataset and then generalizing to unseen examples, Few-Shot Learning (FSL) classifiers learn what makes the elements similar or belonging to the same class from only a few data. Facing the scarcity of data on flaky tests, selecting an FSL classifier seems then to be a promising choice.

In FSL, we call the item we want to classify a *query*, and the *support set* is a small set of data containing few examples for each class used to help the model to make classifications based on similarity as shown in Figure 6.1. To classify flaky tests according to their flakiness category, we compute the similarity between the query and all examples of each flakiness category in our Support Set and assign the label having the maximum similarity with the query. This classification is obviously performed in a space where all elements of the same class are similar or close to each other. This is achieved by a model called *Siamese network*. Its task is to transform the data and project it into a space where all the elements of a same class are close to each other, and then to classify the elements by computing their similarity.

The Siamese network has knowledge of the similarity of elements of the same class. It processes two vectors in input and applies transformations that allow minimizing the distance between the two vectors if they share similar characteristics.

Figure 6.3 shows an example of the visualisation of flaky test vectors before and after the Siamese network is applied. Since CodeBERT has no knowledge of the characteristics of flaky tests and only generates a general representation of the source code, the vectors produced are all similar. However, the Siamese networks learn which characteristics in these vectors are shared by tests of the same class, and thus allow to project vectors into a space that groups tests of the same flakiness category. After this step, it becomes possible to classify them using a similarity computation.

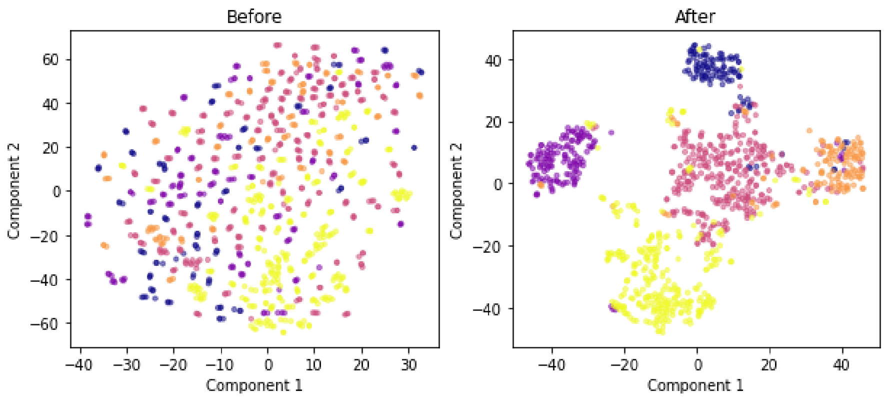


Figure 6.3: Visualisation of our data before and after training of the Siamese network with the triplet loss, which brings together the elements of the same class.

Model training

Siamese networks have two identical sub-networks, each sub-network processes the input vector and performs transformations. Both sub-networks are trained by calculating the similarity between the two inputs and using the similarity difference as a loss function. Accordingly, the weights are adjusted to have a high similarity if the inputs belong to the same class. For the architecture of the sub-networks, we used a dense layer of 512 neurons and a normalisation layer as shown in Figure 6.1. We also performed a linear transformation to keep relations learnt by CodeBERT using the attention mechanism introduced in the transformer architecture [156]. This model is trained using a Triplet Loss function, based on the calculation of similarity difference.

Let the Anchor A be the reference input (it can be any input), the positive example P is an input that has the same class as the Anchor, the negative example N is an input that has a different class than the Anchor, $s()$ is the cosine similarity function, and m is a fixed margin. The idea behind the Triplet Loss function is that we maximize the similarity between A and P, and minimize the similarity between A and N, so ideally $s(A, P)$ is large and $s(A, N)$ is small. The formula for this loss function is:

$$Loss = max(s(A, N) - s(A, P) + m, 0)$$

m is an additional margin as we do not want $s(A, P)$ to be very close to $s(A, N)$, which would lead to a zero loss.

To train the Siamese network with the triplet loss, we give as input batches of pairs with the same classes, and any other pair of a different class can be used as a negative example. We select the closest negative example to the anchor, such as $s(A, N) \simeq s(A, P)$, which generates the largest loss and constitutes a challenge for model learning.

6.3 Interpretability

Model interpretability refers to one's ability to interpret the decisions, recommendations, or in our case the predictions, of a model. Interpretability is a crucial step to increase trust in using a machine learning model. Indeed, it allows model creators to investigate potential biases in the learning processes and better assess the overall performance of their models. On top of that, providing developers with information about how the model came to its prediction can enhance the model adoption [157].

Flakiness prediction approaches often relied on Information Gain to explain what features in the model appeared to be the most useful [92], [96], [152]. In the case of tree-based models, the reported information gain is given by the Gini importance (also known as Mean Decrease in Impurity) [158]. Parry *et al.* [159] used SHapley Additive explanations (SHAP), which is another popular technique for model interpretability [160].

As FlakyCat uses the CodeBERT representation of tests as input, using the previously mentioned techniques would not give understandable features. To our knowledge, there are no existing techniques used for CodeBERT-based model interpretability. Thus, we introduce a novel approach to better understand the decisions of CodeBERT-based models. Following the main motivation of helping developers better understand flaky tests once detected, our goal with this interpretability technique is to arm FlakyCat users with a more fine-grained explanation for the model's decision.

Our technique is inspired by delta debugging algorithms. Delta debugging is used to minimize failure-inducing inputs to a smaller size that still induces the same failure [161]. In our case, we are interested in the particular code statements linked with the most influential information for the model's decision. To identify them, we proceed with the following: We classify all the original test cases and save their similarity scores. We create new versions of each test. Each version is a copy of the original test minus one statement that was removed.

Next, we feed the new versions to FlakyCat. Among all new versions for one test, we keep the one for which the similarity score endured the biggest drop compared to the original prediction score. We consider the statement removed in this version as the most influential one.

6.4 Evaluation

In this section, we explain our evaluation setting for FlakyCat. First, we describe our data curation process, then, we present our approach for answering each of the three research questions.

6.4.1 Data Curation

Collection

For our study, we had to collect a set of flaky tests containing their source code and their flakiness category. We focused our collection efforts on one programming language, as training a classifier using code and tokens from different programming languages is more challenging. For the language choice, we opted for Java, which is the most common language in previous flakiness studies (and thus datasets). To increase the amount of data used in this study, we also collected a new set of flaky tests mined from GitHub that we classified manually.

Table 6.1: Data filtering performed on the different datasets used in this study. Collected represents the new dataset we retrieved.

Filters	**Datasets**				
	[58]	[162]	[163]	[87]	Collected
Inspected commits	201	170	40	101	270
Commit not found	12	12	4	3	3
Duplicated commit	0	2	0	0	3
Open commit	0	0	0	33	0
Flaky test not found	45	21	13	0	42
Configuration problems	3	8	0	0	0
Not Java	15	5	0	0	8
Category hard to classify	40	57	4	0	22
Considered commits	86	65	19	65	192
Total of extracted tests	109	65	20	65	192

Existing datasets There is no large public dataset of flaky tests labelled according to their category of flakiness. Most of the existing studies, such as FlakeFlagger [96] and DeFlaker [83], are limited to list detected flaky tests which are later used for binary classification. Regarding the data classified into flakiness categories defined by Luo *et al.* [58] and Eck *et al.* [49], there is only limited data available in previous empirical studies about flakiness. We retrieved tests from the empirical study of flaky tests across programming languages of Costa *et al.* [162] and from a recent study about pinpointing causes of flakiness by Habchi *et al.* [163]. We also retrieved the flaky tests from iFixFlakies [87] as *Test order dependency* is a flakiness category that received a large interest in the community [59], [88], [159], [164].

We gathered a total of 512 commits/pull requests from the existing datasets we could access, referenced in Table 6.1.

New dataset To expand existing datasets, we explore GitHub projects and search for flakiness-fixing commits for which developers explained the reason (*i.e.* category) of flakiness.

In this search, we use flakiness-related keywords such as *Flaky* and *Intermit* in the commit messages. To ensure that the commit refers to a flakiness category, we further filter commits by specific keywords related to each category: *thread, concurrence, deadlock, race condition* for Concurrency, *time, hour, seconds, date format, local date* for Time, *port, server, network, http, socket* for Network and *rand* for Random. After the search, we rely on the developer's explanation in the commit message and on the provided fix to classify tests into the different flakiness categories listed in the literature. This collection allowed us to obtain 270 commits fixing flaky tests to be classified manually.

Filtering

The previous step allowed us to collect a total of 782 categorized commits/issues. In this step, we filter out commits and data that are not adequate for our study. We filter out commits hard to classify, duplicated ones, and those where flaky tests are not written in java. Costa et al. [162] classified issues, and Luo et al. [58] classified old SVN revisions. In some cases, the corresponding commit could no longer be found in the projects. Some data points were missing necessary attributes, such as the name of the flaky test. Particularly, in commits where the fix is in the production code or in a configuration file, and the test name of the involved flaky test is not indicated in the commit message, we were not able to identify the flaky test, so we filtered them out. The number of tests extracted for each dataset is shown in Table 6.1. The *considered commits* row accounts for commits where all information needed was present *i.e.* the test name, source and category of flakiness. Note that the number of considered commits and extracted tests vary in some cases as developers sometimes addressed more than one flaky test per commit. We obtained a total of 259 flaky tests after filtering the existing datasets. For the data we collected ourselves, we successfully extracted 192 test cases. To ensure the correctness of our manual classification and filtering, the first two authors of the paper performed a double-check on the newly collected dataset.

Processing

After filling in all the necessary attributes: the test case name, flakiness category, test file name, and project URL, we download the code files and extract test methods using the spoon library[2]. At this stage, all comments have been deleted from the source code to restrict CodeBERT to code statements.

Final dataset

The final dataset contains 451 flaky tests distributed over 13 flakiness categories. Table 6.2 illustrates this distribution.

The collected flaky tests are not distributed evenly across categories of flakiness. Just as shown in past empirical studies [58], [61], some categories, such as *Async waits*, are more prevalent than others. Our approach uses FSL to learn from limited datasets. Still, it requires a certain amount of examples to learn common patterns from each category. We decided to have at least 30 tests in a category to consider it. This number is commonly accepted by statisticians as a threshold to have representativeness [165], since learning from very few examples is not feasible. In our dataset, some flakiness categories contain no more than 5 flaky tests. We were not able to gather more data for those non-prevalent categories and thus decided to focus on five of the most common flakiness categories, highlighted in grey in the table: *Async waits*, *Test order dependency*, *Unordered collections*, *Concurrency*, and *Time*.

Data augmentation

Facing the challenge of learning from few data, we over-sampled our training set similarly to SMOTE [166] by applying elementary perturbations. In the same way, as we increase the imagery data by rotating and resizing, for the source code, we generate variants of our tests by mutating only the code elements that have no influence on flakiness. This includes variable names, constants such as strings, test method names, and by adding declarations of unused variables. In this way, the model will learn useful code elements instead of learning from variable names and strings. We used the Spoon library for the detection of these elements, and we replaced them with randomly generated significant words. As a result, the total number of tests after data augmentation is 964.

[2]https://github.com/INRIA/spoon

Table 6.2: Final dataset. The highlighted rows are the data used to train and test the model. The original data refers to the data we collected, short data are tests with less than 512 tokens, and the augmented data are the data we obtained after augmentation.

Class	**Data**		
	Original	**Short**	**Augmented**
Async waits	125	97	300
Test order dependency	103	100	284
Unordered collections	51	48	146
Concurrency	48	40	124
Time	42	38	110
Network	31	25	/
Randomness	17	14	/
Test case timeout	14	9	/
Resource leak	10	7	/
Platform dependency	2	2	/
Too restrictive range	3	2	/
I/O	2	2	/
Floating point operations	3	1	/
Total	**451**	**385**	**964**

6.4.2 Experimental Design

Baseline

To the best of our knowledge, we are the first to introduce an automatic classification of flaky tests according to their category. However, to get a better appreciation of the performance of the solution we propose in this paper, we seek to compare FlakyCat with test representations commonly used by flaky test detection approaches. Our intuition is that test representations giving good performance in binary classification (*i.e.* detecting flaky tests and non-flaky tests) have a good chance to be helpful for the classification of tests according to their category of flakiness. Thus, we use the following representations for our multi-classification task: the vocabulary-based approach [92] which is a keyword-based approach, and the smell-based approach [152] which exploits the correlation between test smells and test flakiness. Our overall motivation is to determine whether it is possible to make this classification based on limited data and to know which combination of classifier and code representation delivers the best results.

For the classification based on test smells, we use the 21 smells detected by tsDetect [167], to generate vectors indicating the presence of each smell detected by the tool, in the same way as in the study of Camara *et al.* [152]. As for the vocabulary-based classification, we use token occurrence vectors, as in the article by Pinto *et al.* [92]. We tokenize the code and apply standard pre-processing like stemming, then calculate occurrences of each token.

In addition to various test representations, we compare our FSL-based approach with traditional classifiers from the Scikit-learn library [168] used by previous studies on flakiness prediction [92], [95], [152]: Random Forest (RF), Support Vector Machine (SVM), Decision Tree (DT) and K-Nearest Neighbour (KNN).

To validate each model, we split our data into 75% for training and 25% for final validation. We use a 10-fold stratified cross-validation on the training data to select the best model parameters and use those parameters to evaluate the model on the unseen hold-out set.

As the augmented samples in our dataset are variants of the original ones, it was important to keep them in the same sets, to ensure that no similar data pairs are included in both the training and test sets. For the support set used for classification, we select the most centred examples to represent each class.

FlakyCat relies on a Siamese network. It is trained with combinations of data by indicating whether these data are similar or not so that the model can learn what makes them similar. Since we train with combined data, the balancing of data is not required, because it is automatically over-sampled.

Parameters

We tuned FlakyCat's parameters on the training set using the Random Search method [169] and a 10-fold cross-validation, by testing random combinations of the most important parameters that have a direct impact on the model performance, which include the similarity margin used in the triplet loss function, the learning rate, the number of warm-up steps, and the support set size.

Figure 6.4 shows the resulting weighted F1 score for each tested parameter combination using a 10-folds cross-validation. A high learning rate and a number of warm-up steps have a negative impact on the performances, while other parameters have a lower influence. Following these results, for the final validation on the hold-out set, we use the best parameter combination identified in the Figure 6.4: a similarity margin of 0.30, a learning rate of 0.001, a number of warm-up steps of 400, and a support set with 10 examples from each category.

For baseline classifiers, we keep the standard values used by previous works. We varied the number of trees in the Random Forest classifier, we tested values from 100 to 1000 with a step of 100. We observed that this does not make much difference regarding the F1 score ($\leq 3\%$), and we identified 1000 as the number giving the best results.

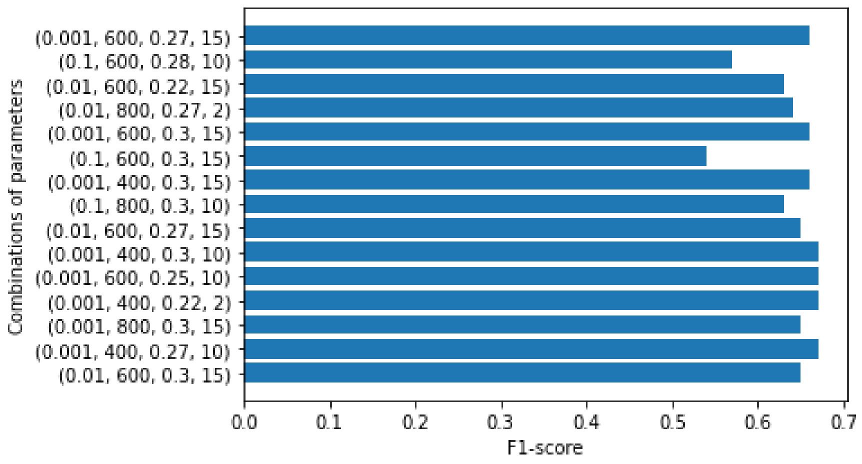


Figure 6.4: F1 score for different values of parameter combinations using Random Search and a 10-Folds cross-validation. The combinations on the Y axis have the form : (learning rate, number of warm-up steps, similarity margin, support set size).

Evaluation metrics

We use the standard evaluation metrics to compare classifiers, including precision, recall, Matthews correlation coefficient (MCC), F1 score, and Area under the ROC curve (AUC). These metrics have been used to evaluate the performance of classifiers, including binary classification of flaky tests [92], [97], [152]. Since our dataset is unbalanced, weighted metrics are more suitable for our evaluation.

Research questions

RQ1: *How effective is FlakyCat compared to approaches based on other combinations of test representation and classifier?*
This question aims to evaluate FlakyCat and compare it to other test representation techniques, *i.e.* vocabulary and test-smell-based and traditional classifiers, *i.e.* SVM, KNN, decision tree and random forest.

RQ2: *How effective is FlakyCat in predicting each of the considered flakiness categories?*
This question evaluates FlakyCat's ability to classify the different categories of flakiness. To perform this, we split the dataset into five sets following the categories: *Async waits*, *Test order dependency*, *Unordered collections*, *Concurrency*, and *Time*. Then, we use the same settings as for RQ1 to tune the Siamese network, train it, and evaluate it for each category.

RQ3: *How do statements of the test code influence the predictions of FlakyCat?*
We applied the technique we introduced in Section 6.3 for CodeBERT-based model interpretability to FlakyCat. We classified all original short data (323 tests). For 16 tests, the score doesn't decrease by deleting one statement, and thus we collected 307 statements of interest, important for FlakyCat's decision-making. To better understand what information emerges, we proceeded with the following analysis. First, we regroup statements by category of flakiness (according to the flaky test they belong to). Then, we want to share information on what type of statements FlakyCat found useful. To do so, we look through the list of statements and attempt to identify recurring code statements and categorize them. The process of identifying statement types is exploratory and inspired by qualitative research. Two of the authors of this paper went through the list of statements and identified nine recurring types of statements:

- **Control flow:** Includes decision-making statements, looping statements, branching statements, Exception handling statements.

- **Asserts:** All types of assertions in tests.
- **Threads:** statements related to threads and runnables.
- **Constants:** Constant values such as strings, numbers and boolean values independent of variables, and final variables.
- **Waits:** All explicit wait statements.
- **Usage of date/time:** Statements that perform operations on time values, dates.
- **Network:** Statements related to data exchange in a local or external network between two endpoints, and session management.
- **I/O:** Statements related to input/output, database and file access.
- **Global variables:** Includes the use of global variables.

With this question, we investigate the prevalence of these statement types in each flakiness category.

6.5 Results

6.5.1 RQ1: How effective is FlakyCat compared to approaches based on other combinations of test representation and classifier?

Table 6.3: Comparing performances of FlakyCat (CodeBERT and Few-Shot Learning) with traditional machine learning classifiers

Model	Smells-based					Vocabulary-based					CodeBERT-based				
	Precision	Recall	MCC	F1	AUC	Precision	Recall	MCC	F1	AUC	Precision	Recall	MCC	F1	AUC
SVM	0.11	0.34	0.00	0.17	0.50	0.61	0.52	0.37	0.45	0.66	0.27	0.43	0.22	0.33	0.60
KNN	0.24	0.37	0.11	0.29	0.55	0.44	0.48	0.31	0.45	0.65	0.56	0.53	0.37	0.51	0.68
DT	0.31	0.33	0.10	0.23	0.53	0.53	0.53	0.39	0.52	0.69	0.49	0.50	0.34	0.49	0.67
RF	0.32	0.34	0.12	0.24	0.54	0.72	0.61	0.49	0.56	0.72	0.68	0.66	0.55	0.62	0.76
FSL	0.13	0.18	-0.01	0.13	0.50	0.69	0.68	0.58	0.67	0.79	**0.74**	**0.73**	**0.65**	**0.73**	**0.83**

Following the outlined experimental design, we trained and tested FlakyCat and the four traditional classifiers, using the three source code representations, the vectors obtained from CodeBERT, the vectors based on vocabulary, and the ones based on test smells. The obtained results are presented in Table 6.3. The results show that FlakyCat achieves the best performance for all evaluation metrics. It obtained an average weighted F1 score of 73% and a precision of 74%. We get an MCC of 0.65 (bounds for this metric are between -1 and 1), being close to 1 means a perfect classification. Finally, the AUC of 0.83 shows that the model is able to distinguish flaky tests from different classes.

Representation effect Regarding the three code representations, CodeBERT achieves the best performance for RF, KNN, and FSL, with an F1 score between

0.51 and 0.73 for the three classifiers. When using the vocabulary-based vectors, SVM and DT perform better than using CodeBERT. With this representation, all classifiers do not exceed an F1 score of 0.67. The representation based on test smells yields lower results, with the best F1 score being 0.29. The CodeBERT representation seems then promising to use when learning to classify flaky tests according to their categories.

Classifier effect Regarding the choice of classifier, we find that the FSL classifier based on similarity achieves the best performance using the representations based on CodeBERT and vocabulary. Among traditional classifiers, Random Forest obtains the best results, as reported in previous flaky test classification studies [92], [94]. Classifiers relying on the smell-based representation have more difficulty to classify flaky tests. Using this code representation, the KNN classifier achieved the best F1 score: 0.29. Two categories had a positive impact to achieve this score: *Async wait*, and *Test order dependency*. This can be explained by the presence of test smells strongly related to these two categories, including Sleepy test and Resource optimism. Other flakiness categories seem to be more challenging to predict using existing test smells.

Random-guessing comparison In the previous paragraph, we compared different models and different code representations and saw that FlakyCat gave the best results. Because all the existing approaches were designed to detect flaky tests from non-flaky tests, they might not be suitable for the specific task of classifying flaky tests according to their categories. As no other category-based classification technique exist so far, we show the performance of a random guesser as another baseline. We consider two random guessing approaches, the first one where we randomly affect a class to each flaky test and the second one where we weigh the random affectation according to the prevalence of flaky tests in each category. Both approaches are considered as the dataset balance might be different from the one found in various projects. Results are listed in Table 6.4. F1 scores for Random and Weighted Random are respectively of 0.21 and 0.25. With an F1 score of 0.73, we see that FlakyCat performs better than the two considered random-guessing approaches.

RQ1: Overall, our results show that automatic classification of flaky test categories with a limited amount of data is a challenging but feasible and promising task.
The representation based on CodeBERT gives better results compared to the ones based on test smells and vocabulary. We also found that Few Shot Learning performs better than traditional machine learning classifiers.

Table 6.4: Performance of random guessing approach

Method	Precision	Recall	MCC	F1	AUC
Random	0.25	0.20	-0.01	0.21	0.50
Weighted Random	0.25	0.26	0.02	0.25	0.51

6.5.2 RQ2: How effective is FlakyCat in predicting each of the considered flakiness categories?

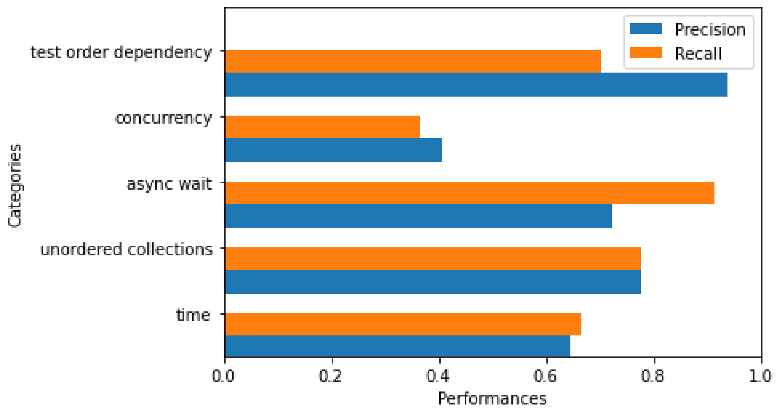


Figure 6.5: Precision and Recall per flakiness category using FlakyCat

Figure 6.5 shows performances achieved by FlakyCat for each of the five flakiness categories. Results show that the category *Async wait* is the easiest for the model to classify, with an F1 score of 0.81. The category *Test order dependency*, *Unordered collections* and *Time* respectively have an F1 score of 0.80, 0.78 and 0.66. *Concurrency* performances are lower with an F1 score of 0.39. We suspect that concurrency issues happen in many cases in the code under test. As FlakyCat only relies on the test source code, this would indeed explain why performances are lower in this case. Another supposition is that concurrency issues and asynchronous waits are sometimes closely related. We discuss an example of this in Section 6.6.1.

RQ2: While the four flakiness categories *Async waits*, *Test order dependency*, *Unordered collections*, and *Time* show good ability to be detected automatically, *Concurrency* remains difficult to detect by relying only on the test case code.

Table 6.5: Prevalence of different types of statements in each flakiness category for true positive predictions

	#Statements	Control flow	Constants	Asserts	Threads	Waits	Network	Global variables	Usage of Date/time	I/O
Async Waits	80	16,25%	55%	20%	20%	27,5%	25%	11,25%	3,75%	6,25%
Concurrency	34	23,53%	47,06%	17,65%	29,41%	14,70%	14,70%	5,88%	17,65%	2,94%
Test order dependency	69	8,69%	60,87%	13,04%	0%	4,35%	8,69%	2,90%	7,25%	47,82%
Time	32	18,75%	56,25%	50%	0%	0%	0%	9,375%	62,5%	6,25%
Unordered collections	42	4,76%	66,67%	38,09%	0%	0%	2,38%	4,76%	0%	4,76%

6.5.3 RQ3: How do statements of the test code influence the predictions of FlakyCat?

Table 6.5 reports the prevalence (%) of the different types of statements among all influential statements per flakiness category, *e.g.* 100% Asserts in the *Time* category would mean that all influential statements for the *Time* category contain assert statements.

Compared to other flakiness categories, the percentage of assertions in the influential statements of *Time* and *Unordered collections* is high, 50% and 38.09% respectively. Based on our analysis, this includes in particular assertions that perform exact comparisons, such as `assertEquals()`, between constant values and collection items, or dates for example. 29,41% of influential statements in the *Concurrency* category include thread manipulation, and 20% for the *Async Waits* category, while the rest of the categories have none. Statements containing explicit waits represent respectively 27,5% and 14,7% for *Async Waits* and *Concurrency* categories, but below 5% for *Test order dependency* and zero for the others. Statements containing date or time values are most common in the *Time* category with 62,5%. We note that they appear as well in a small proportion, 17,65%, for *Concurrency*. Statements from the I/O calls group are mainly found in the *Test order dependency*. For *Control flow*, *Constants*, and *Global variables* statements are almost evenly distributed. We include a spreadsheet containing all statements analyzed in our replication package.

RQ3: The interpretability technique we presented enable us to find which statements impact FlakyCat's decision. We also find hints that specific flakiness categories have distinct statement types (*e.g.* Usage of Date/time for the *Time* category) while some others have similar prevalence (*e.g.* Threads for *Async Waits* and *Concurrency* categories). By highlighting these statements, our interpretability technique may provide information to developers to better understand flaky tests, their categories and their causes.

6.6 Discussion

6.6.1 Reasoning about the statements influencing FlakyCat and the usage of flakiness categories

Listing 6.1 gives an example of a flaky test taken from the Neo4J project[3] found during the data collection part. As explained in the commit message, the flakiness was caused by a race condition and thus, we affected it to the Concurrency category. FlakyCat classified this test as Async wait. The interpretability technique that we introduced in Section 6.3 reveals that the statement on line 6 is the most influential for the model's decision. It contains the `await()` function, and this is likely the reason why the flaky test was categorized as Async wait. Furthermore, similarity score for the Concurrency category is high, and it comes as FlakyCat's second guess.

When looking at the test, we understand that an asynchronous wait was performed to wait for a thread. We also found similar examples concerning other categories, such as waits relying on network resources. First, we argue that our interpretability technique can help to understand the cause of flakiness, even when FlakyCat apparently mislabelled the test. Secondly, we advance that flakiness categories as commonly defined in research studies [49], [58] can overlap, *i.e.* a flaky test can belong to several categories. The application of machine learning to determine the causes of flakiness is promising and should receive attention. It would also benefit from a more precise, orthogonal classification of flakiness categories.

Listing 6.1: A flaky test belonging to two categories

```
 1 @Test
 2 public void shouldPickANewServer[...]() throws Throwable {
 3 [...]
 4     Thread thread = new Thread( () -> {
 5     try {
 6     startTheLeaderSwitching.await();
 7     CoreClusterMember theLeader = cluster.
 8         awaitLeader();
 9     switchLeader( theLeader );
10     } catch ( TimeoutException | InterruptedException e ) {
11         // ignore
12     }});
13 [...]
14 }
```

[3]https://github.com/neo4j/neo4j/commit/c77e579b40b02087

6.6.2 The effect of considering an additional category

Our results showed that flakiness categories can be classified automatically. We carried out our main experiments with five categories of flakiness for which we had a reasonable number of tests. Still, we believe that one interesting aspect of our study is to understand the impact of adding other categories to FlakyCat. For this, we investigate the performance of FlakyCat for each category (similarly to RQ2), but we add to our set the *Network* category, which is the next category with the most samples in our dataset (25 tests). F1 scores and the accuracy obtained for each category are presented in Figure 6.6.

Compared to the results previously reported in Table 6.3, we observe that the performances of each category are slightly impacted. The *Async waits* category is the most impacted one. Indeed, after adding the *Network* category, we get an overall F1 score of 0.68. The added category gets the worst results. This performance drop can be explained by multiple factors. First, having more categories to differentiate makes it more challenging for FlakyCat to distinguish between them. Secondly, the overall F1 score is strongly affected by the poor performance observed in the new category. The performance for the Network category can be a result of the too low number of examples in this category (25). Despite using FSL, the model still requires enough data points in each category. While collecting data, we noticed that flaky tests caused by Network were not common. These findings align with the ones about the prevalence of the different categories reported in previous empirical studies [49], [58]. In addition, flaky tests related to *Network* issues could also be considered as Asynchronous waits in many cases, as previously explained.

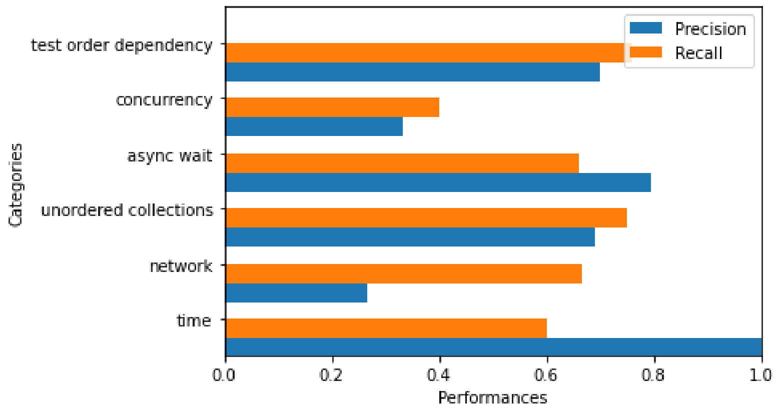


Figure 6.6: Precision and Recall per flakiness category when adding the category "Network"

6.7 Threats to Validity

6.7.1 Internal Validity

One threat to the internal validity is related to the dataset we used in our study. Flaky tests were gathered from different sources, as explained in Section 6.4.1. It is possible that flaky tests were assigned to the wrong label, which would impact the training and evaluation of our model. Certifying the category based on the test source code is complex and can as well be subjective. To ensure the quality of the data, the first two authors reviewed the collected flaky tests and confirmed their belonging to the assigned category.

Similarly, the identification of statement types in RQ3 required a manual analysis of the most influential statements. Hence, the identified types can be subjective and the assignment of statements is prone to human errors. To mitigate this risk, we kept the statement types factual, *e.g.* control flow and asserts. This allows us to avoid assignment ambiguities and intersections between the different statement types.

6.7.2 External Validity

The first threat to external validity is the generalizability of our approach. In this study, we train a model to recognize flaky tests from four of the most prevalent categories, but we are not sure of the performances in other categories. We discussed the addition of two categories (Network and Randomness) and retrieved that the number of examples is one of the influencing factors.

6.7.3 Construct Validity

One potential threat to construct validity regards the metrics used for the evaluation study. To alleviate this threat, we report MCC, F1 score, and AUC metrics in addition to the commonly-used precision and recall. As our data is not evenly distributed across the different categories, we report the weighted F1 score.

6.8 Conclusions

Test flakiness is considered as a major issue in software testing as it disrupts CI pipelines and breaks trust in regression testing.

Detecting flaky tests is resourceful, as it can require many reruns to reproduce failures. To facilitate the detection, more and more studies suggest static and dynamic approaches to predict if a test is flaky or not. However, detecting flaky tests constitutes only a part of the challenge, since it remains difficult for developers to understand the root causes of flakiness. Such understanding is vital for addressing the problem, *i.e.* fixing the cause of flakiness. At the same time, researchers would gain more insights based on this information. So far, only a few automated fixing approaches were suggested and these are focusing on one category of flakiness. Knowing the category of flakiness for a given flaky test is thus a piece of key information.

With our work, we propose a new approach to this problem that aims at classifying previously identified flaky tests into their corresponding category. We propose FlakyCat, a Siamese network-based multi-class classifier that relies on CodeBERT's code representation. FlakyCat addresses the problem of data scarcity in the field of flakiness by leveraging the Few-Shot Learning capabilities of Siamese networks to allow the learning of flakiness categories from small sets of flaky tests. As part of our evaluation of FlakyCat, we collect and make available a dataset of 451 flaky tests with information about their flakiness categories.

Our empirical evaluation shows that FlakyCat performs the best compared to other code representations and traditional classification models used by previous flakiness prediction studies. In particular, we reach an F1 score of 73%. We also analyzed the performances with respect to each category of flakiness, showing that flaky tests belonging to *Async waits*, *Test order dependency*, *Unordered collections*, and *Time* are the easiest to classify, whereas flaky tests from the *Concurrency* category are more challenging. Finally, we present a new technique to explain CodeBERT-based machine learning models. This technique helps in explaining what code elements are learnt by models and give more information to developers who wish to understand flakiness's root causes.

7 Pinpointing Classes Responsible for Test Flakiness

Still with the goal of helping to debug flakiness, we present in this chapter a new approach to locate the source of flakiness when it originates from the code under test. This represents a specific but critical case as the cause of non-determinism lies in the program itself. The approach leverages Spectrum-Based Fault Localisation techniques, code and change metrics and ensemble learning to rank classes the most likely to be responsible for flakiness.

This chapter is based on the work published in the following paper:

- S. Habchi, G. Haben, J. Sohn, *et al.*, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 352–363. DOI: `10.1109/ICSME55016.2022.00039`

Contents

7.1 Introduction

Regression testing is a key component of continuous integration (CI) that checks whether code changes integrate well in the codebase without breaking any existing functionality. To this end, it is assumed that failing tests indicate the presence of faults, introduced by the latest changes. However, some tests break this assumption by failing for reasons other than faults, as for instance, they exhibit non-deterministic behaviour, thereby sending confusing signals to developers. Such tests are usually called *flaky tests*.

Academic and industrial reports have emphasised the adverse effects of test flakiness in software development. Specifically, Google reported that 16% of their tests manifested some level of flakiness, while more than 90% of their test transitions, either to failing or passing, were due to flakiness [57]. As the *de facto* approach for detecting flaky tests is to rerun them [81], [105], detecting large numbers of flaky tests can be time- and resource-consuming. Indeed, Google reports that between 2 to 16% of their CI resources are dedicated in rerunning flaky tests [20]. It is noted that other companies, like Microsoft [141], Spotify [54] and Mozilla [170], also report similar issues when dealing with test flakiness.

Perhaps more importantly, test flakiness affects team productivity and software quality [105]. This is because flaky failures interrupt the CI and make developers waste time in investigating false issues [49], [56], [57], [105]. Additionally, the accrual of flaky tests breaks the trust in regression testing, leading developers to disregard legitimate failure signals believing them to be false [81], [105]. This situation often results in faults slipping into production systems [170]. Moreover, code quality is often linked with the level of flakiness incurred [105] and thus, developers need to know where it comes from and understand the causes of flakiness to avoid introducing and spreading it.

Given the adverse effects of test flakiness, engineers and researchers aim at developing

detection techniques that can predict whether a test is potentially flaky. These approaches rely on a number of runs and re-runs, such as iDFlakies [59] and Shaker [84], coverage analysis like DeFlaker [83], or static and dynamic test features [92], [94]–[97], [152], [171]. Evaluated on open-source projects, these approaches showed promising detection accuracy and considerably decreased the amount of time and resources needed to detect flaky tests.

Although flakiness detection methods are important, alone, they cannot reduce the prevalence of test flakiness. This is because on the one hand there are only partial approaches to the problem, such as iFixFlakies [87] and Flex [172] that are only applicable to specific cases, and the inherent difficulties in isolating/controlling the flakiness causes on the other. For instance, iFixFlakies [87] fixes order-dependent tests by identifying helper statements in other tests, whereas Flex [172] identifies assertion bounds that minimise flakiness stemming from algorithmic randomness. At the same time, many prevalent categories of flakiness, *e.g.* Asynchronous Waits and Concurrency [49], [58], [61], [76], remain unaddressed by fixing approaches. This is mainly due to the difficulty of identifying and controlling the cause of flakiness [49].

Flakiness root cause localisation is both important and difficult. It is important since it allows developers to understand the sources of flakiness, hence enabling better control of non-determinism. It is also difficult because of the difficulty to reproduce failures, the diversity in potential issues, *e.g.* time and network, and the large scope of potential culprits, *e.g.* the tests, the code under test (CUT), and the infrastructure [61]. Consequently, practitioners struggle to identify the causes of non-determinism in their codebases that trigger flakiness and consider this step as the main challenge in automating flakiness mitigation strategies [49].

In this paper, we address this challenge by re-targeting Fault Localisation (FL) techniques in order to help identify components (program classes in particular) that are responsible for the non-deterministic behaviour of flaky tests. For the sake of simplicity, we refer to these classes as *flaky classes*. Such techniques can be useful to support the analysis of codebases and of flaky tests. Thus, given a failure, either known as flaky or unknown, engineers can rely on localisation methods to investigate the specific scenario (condition) that causes the test transition. Additionally, flakiness localisation techniques can help with code comprehension and make engineers aware of code areas linked with flaky behaviour, assisting them in both development and testing tasks.

In view of this, we investigate the appropriateness of a variety of fault localisation methods, such as Spectrum-Based Fault Localisation (SBFL), change history metrics, and static code metrics in identifying flaky classes. Our study aims to answer the following four research questions:

RQ1: *Are SBFL-based approaches effective in identifying flaky classes?*

RQ2: *How do code and change metrics contribute to the identification of flaky classes?*

RQ3: *How can ensemble learning improve the identification of flaky classes?*

RQ4: *How does an SBFL-based approach perform for different flakiness categories?*

To answer these questions, we analyse five Open Source projects where test flakiness has been fixed during the project evolution. Our analysis shows that:

- An ensemble of models based on SBFL, change, and size metrics, yields the best results, with 61% of flaky classes in the top 10 and 26% of them at the top. This method also reduces the average effort wasted by developers to 19% of the effort spent when inspecting all classes covered by the flaky test.
- The ensemble method is effective for major flakiness categories. Concurrency

Table 7.1: Collected Data. *ffc:* number of flakiness-fixing commits. *all:* number of commits in the project.

Proj.	**#Commits**		**#Tests**		**#Classes**	
	ffc	all	min - max	avg	min – max	avg
Hbase	8	18,990	138 - 2,089	1,113	734 – 1366	1053.4
Ignite	14	27,903	15 - 1,018	174	72 – 1767	1262.3
Pulsar	10	8,516	194 - 1,326	626	171 – 422	259.7
Alluxio	3	32,560	315 - 694	473	131 – 817	360.3
Neo4j	3	71,824	21 - 5,782	2,139	40 – 1663	581.3
Total	38		15 - 5,782	905	40 – 1767	820.2

and Asynchronous Waits are identified effectively, with 38% and 30% of their flaky classes ranked at the top, respectively.

To facilitate the reproducibility of this study, we provide all used scripts, the set of collected flaky classes, and detailed results in a comprehensive package[1].

7.2 Data Collection

The objective of our study is to assess the effectiveness of FL techniques in identifying flaky classes. To achieve this, we need a set of flaky tests for which the responsible classes are already known. For this, we rely on flakiness-fixing commits as they provide information about classes that were modified as part of the fix. Our assumption is that such classes are, at least, part of the root cause. To collect flaky classes, we followed a four-step process.

Search This step aims to identify Java projects containing the highest number of flakiness-fixing commits. For this, we relied on two sets of projects to consider. We built the first set by using the SEART GitHub Search Engine [173]. Out of the 81,180 available Java projects, we selected the top 200 projects for each of those criteria: number of commits, contributors, stars, releases, issues, and files. This sorting was made with the aim of finding the bigger and more complex projects, thus maximising our chance to find flakiness-fixing commits. Keeping only unique projects in those sets, we ended up with a first list of 902 projects. As a second set, we use the 187 projects available in the iDFlakies dataset [59]. For each of the 1,089 projects, 902 from the first and 187 from the second set, we query the GitHub API looking for commits with messages containing the keyword *flaky*. This led to the identification of 16,501 commits. We look further into whether these commits are truly suitable for our purpose through the following processes.

Inspection The objective of this step is to filter commits that do not provide a clear indication about the flaky class. Hence, we look for flakiness-fixing commits

[1] `https://github.com/serval-uni-lu/sherlock.replication`

containing any of the following keywords: *fix, repair, solve, correct, patch, prevent.* Then, we analyse each commit and keep the ones that:

- The fix affects the code under test (not only the test itself);
- The changes are atomic enough (*i.e.* containing only relevant changes) allowing us to discern the flaky class(es).

This led to the selection of 85 commits from five projects. We further discarded 22 commits for which the flaky tests or commits were not retrievable (*e.g.* rejected pull request), leaving 63 commits in the end.

Test execution This step aims to select commits that are usable in our evaluation. Our first question inspects the effectiveness of SBFL, a technique that requires a coverage matrix indicating the classes covered by each test. Hence, for a commit to be usable in our analysis, its test suite should be runnable allowing us to extract the coverage matrix. To ensure this, we used GZoltar[2], a Maven plugin that allows collecting coverage information for each commit.

For 11 commits, we were unable to run GZoltar due to an incompatible Java version. We also found that the flakiness patches were irrelevant in 10 commits. For instance, some commits were fixing modules in other programming languages or modifying non-source code files. Lastly, we filtered out four additional commits since the reported flaky failures were not *flaky test failures.* Consequently, we dropped 25 commits in addition. Table 7.1 summarises the retained projects. The complete list of flakiness-fixing commits is available in our replication package.

Extraction For each collected flakiness-fixing commit, we retrieve the source code, the test suite, the fixed flaky test, and the flaky class. To retrieve the flaky classes, two authors manually analysed the commit diff and message to identify them. Overall, the identification was obvious since we selected atomic commits beforehand. Hence, there were no disagreements between the authors at this step. The identified classes are considered the ground truth of our study.

7.3 Study Design

7.3.1 RQ1: Effectiveness

Motivation

The objective of our study is to investigate the usability of well-founded FL techniques to help in mitigating flaky tests. The literature on FL proposes a wide variety of categories such as ML-based techniques [174]–[176], mutation-based techniques [177], [178], and qualitative reasoning-based techniques [179]. Nonetheless, spectrum-based fault localisation remains one of the most distinguished FL categories thanks to its effectiveness and simplicity [180].

[2]`https://github.com/GZoltar/gzoltar/blob/master/com.gzoltar.ant/`

SBFL requires only the test coverage matrix to compute the likelihood for a code entity to include the root cause of an observed test failure. The main assumption of SBFL is that code entities covered by more failing tests and fewer passing tests are more suspicious than those less covered by failing tests and more by passing tests [181]. This assumption can be revised to identify the root causes of flaky tests instead of bugs. In particular, if we separate tests into two groups: *flaky* and *stable*, instead of *failing* and *passing*, we can leverage the coverage matrix to rank classes based on their correlation with flaky tests. In this case, the assumption would be that classes covered by more flaky tests and fewer stable tests have a higher chance to be responsible for test flakiness. In this RQ, we assess the effectiveness of this adaptation of SBFL in identifying flaky classes.

Approach

Relying on the data collected in Section 7.2, we use the GZoltar plugin to run the test suites of each commit and build coverage matrices. Based on these matrices, we compute for each class the spectrum data: (e_s, e_f, n_s, n_f). In our case, for each class, e_s and e_f represent the number of stable and flaky tests executing it, respectively. On the other hand, n_s and n_f represent the number of stable and flaky tests that do not execute it, respectively. To compute classes' suspiciousness scores, we inject these spectrum data in classical SBFL formulæ. Table 7.2 summarises the four formulæ adopted in our study with the necessary adaptations for flakiness. For DStar, the notation '*' is a variable that we set to 2 based on the recommendation of Wong *et al.* [72]. With each formula, we compute the suspiciousness scores of each class and then rank them in descending order: classes with the highest scores are ranked first.

Recently, it has been theoretically proven that no SBFL formula can outperform all others [182]. In addition, Xuan and Monperrus proposed a new approach that learns to combine multiple SBFL formulæ [183]. Their approach, called Multric, successfully outperformed all the input formulæ, opening a trend to use multiple formulæ to overcome the limitation of using a single SBFL formula [175], [184], [185]. Following this trend, we used Genetic Programming to evolve a new formula that combines all four SBFL formulæ.

Genetic Programming (GP) evolves a solution (*i.e.* a program) for a given problem under the guidance of a (fitness) function. GP can also generate non-linear models and learn a model flexibly from input instead of defining a fixed formula. Hence, GP was employed to generate risk evaluation formulæ for fault localisation [186], [187]. For the same reasons, we employ GP to evolve a model (*i.e.* a formula) for the flaky class identification problem. We configure the GP to have a population of 40 individuals and to stop and return the best model found so far after 100 generations. Each individual in the population denotes a single candidate formula and is generated using (i) six arithmetic operators (subtraction, addition,

Table 7.2: SBFL formulae adapted to flakiness.

Name	**Formula**
Ochiai [71]	$\frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_s)}}$
Barinel [73]	$1 - \frac{e_s}{e_s+e_f}$
Tarantula [189], [190]	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_s}{e_s+n_s}}$
DStar [72]	$\frac{e_f^*}{e_s*n_f}$

multiplication, division, square root, and negation) and (ii) the features that GP takes as input. We define our fitness function as the average ranking of flaky classes. To make most of the data and avoid overfitting, we use ten-fold cross-validation, using one fold for test and the others for training. We also normalise all input data between 0 and 1 using min-max normalisation. Finally, to compensate for the inherently stochastic nature of GP, we run GP 30 times with different random seeds and report the results of a model with the median fitness. We used DEAP v.1.3.1 [188].

7.3.2 RQ2: Code and Change Metrics

Motivation

The objective of this question is to explore the benefits of augmenting the SBFL technique with additional signals from the software. Recent studies showed that the performances of SBFL can be improved by incorporating signals from code and change metrics. More specifically, Sohn and Yoo [187] showed that combining SBFL with code and change metrics widely adopted in the fault prediction community [191], such as age, change frequency (*i.e.* churn), and size, can significantly improve the approach's performances.

The assumption is that code entities with higher complexity and change frequency are more likely to be faulty. Several studies suggested that the test size and complexity can also be an indicator of flakiness [91], [92], [152]. However, it is unclear if such metrics correlate also with classes that are responsible for test flakiness. Therefore, in this RQ, we assess the benefits of these metrics in spotting flaky classes. Besides these metrics, we investigate the effects of metrics that are specific to the nature of flaky tests. Multiple empirical studies analysed the root causes of flakiness and showed that the main categories are: Async Waits, Concurrency, Order-dependency, Network, Time, I/O operations, Unordered collections and Randomness [58], [61], [62], [74]. We derived a list of static metrics that describe each of these categories in Java projects. We exclude order-dependency

Table 7.3: Code and change metrics used to augment SBFL.

	Metric	Definition
Flakiness	#TOPS	Number of time operations performed by the class.
	#ROPS	Number of calls to the `random()` method in the class.
	#IOPS	Number of input/output operations performed by the class.
	#UOPS	Number of operations performed on unordered collections by the class.
	#AOPS	Number of asynchronous waits in the class.
	#COPS	Number of concurrent calls in the class.
	#NOPS	Number of network calls in the class.
Change	Changes	Number of unique changes made on the class.
	Age	Time interval to the last changes made on the class.
	Developers	Number of developers contributing to the class.
Size	LOC	The number of lines of code.
	CC	Cyclomatic complexity.
	DOI	Depth of inheritance.

because order-dependent tests generally stem from tests themselves instead of the CUT, thus, they are not concerned by our approach. In the following, we describe our approach for (i) calculating these metrics and (ii) defining an FL formula based on them.

Approach

Metric collection Table 7.3 summarises the full list of metrics used in our study. To compute these metrics, we first retrieve the source code of the project at the commit of interest (*i.e.* the parent commit of the flakiness-fixing commit identified by the data collection step). Then, for calculating flakiness-specific metrics, we use Spoon [192]. Spoon is a framework for Java-based program analysis and transformation that allows us to build an abstract syntax tree and a call graph. Using the graph and tree, we extract classes and their metrics (*e.g.* #COPS and #ROPS). For size metrics, we also use these code analysis results from Spoon (*e.g.* DOI). As for change metrics, we analyse the change history and extract the following information: the date of each commit, files modified and renamed by each commit, and authors of individual commits. Using this information, we compute the three change metrics: Unique Changes, Age, and Developers.

Ranking model Similarly to RQ1, we use GP in order to generate models that combine our metrics with suspiciousness scores generated by SBFL formulæ. In particular, for each type of metrics (*i.e.* flakiness, size, and change), we evolve a model that takes as input its metrics with SBFL scores and outputs a ranking for each candidate class. Afterwards, we compare the performances of these models to infer the contribution of each type of metrics.

7.3.3 RQ3: Ensemble Method

Motivation

This question explores the potential for improvement by exploiting all the formulæ generated using GP while at the same time making the most of the resources spent on model generation. For this aim, we use voting as our ensemble learning method. We opted for voting since it does not require an additional cost for model generation and its effectiveness has already been demonstrated by previous fault localisation studies [193], [194]

Approach

Voting between models is performed in two phases: candidate selection and voting. During the candidate selection phase, all the participating models compute their own suspiciousness scores for the candidates. A candidate, in our case, is an individual class of the CUT. Individual models compute their own suspiciousness scores for the candidates and select those placed within the top N as their candidates to vote. In the voting phase, each model votes for its own top N candidates. If M number of models participating in the voting, we can have the maximum $N \times M$ number of voted candidates in total. The votes from the models are then aggregated, and the voted candidates are reordered from the most voted to the least voted.

Previous studies on voting-based FL showed that varying the number of votes that each candidate receives based on its actual rank in individual models can improve the localisation performance even further [193], [194]. Hence, rather than assigning the same number of votes to each candidate, we allow individual models to cast a different number of votes for each candidate based on its location in the ranking. For instance, a candidate ranked at the top will obtain a complete one vote, whereas a candidate ranked in the third place will get $\frac{1}{3}$ vote. As mentioned in 7.3.6, candidates can be tied with other candidates since their ranks are computed from ordinal scores. When a candidate fails to be in the top N due to being tied with others, we allow every tied candidate (c) to receive the following number of votes: $votes = \frac{1}{rank_{best}(c) \times n_{tied}(c)}$ votes. Here $rank_{best}$ denotes the best (highest) rank a tied candidate can have, and n_{tied} is the total number of tied candidates, including itself. The equation below summarises the number of votes a candidate (c) can obtain. $rank(c)$ is the rank of the candidate c.

$$\begin{cases} \frac{1}{rank(c)} & \text{if } rank(c) \leq N \\ \frac{1}{rank_{best}(c) \times n_{tied}(c)} & \text{if } rank_{best}(c) \leq N \\ 0 & \text{otherwise} \end{cases}$$

7.3.4 RQ4: Flakiness Categories

Motivation

The literature on flaky tests reports different categories of flakiness [58], [61], [62], [74]. These categories can manifest differently both in the test and CUT and as a result the identification of flaky classes can also be affected by such differences. That is, a technique might identify decently the classes responsible for non-deterministic network operation, but struggles in pinpointing classes causing race conditions. This RQ aims to investigate the performances of an SBFL-based approach among distinct flakiness categories.

Approach

Many studies manually analysed flakiness-fixing commits to categorise them [58], [75] based on their commit message and code changes. In our study, we followed a similar process where two authors manually analysed the commits separately to assign them to one of the categories derived by Luo *et al.* [58]. As our manual analysis does not intend to build a new taxonomy or identify new categories, it is reasonable to adopt an existing taxonomy as a reference. The two authors had a disagreement over one commit, where one author only suggested one category whereas the other suggested two categories. After discussion, the authors decided to keep two categories to avoid discarding relevant information. The results of this analysis are available in our replication package. After labelling the flakiness-fixing commits, we analyse the performance of our SBFL-based approach among different flakiness categories.

7.3.5 Evaluation metrics

For the evaluation of our approach, we use two metrics: accuracy and wasted effort. Both *acc@n* and *wef* are based on the absolute number of code entities instead of percentages. This conforms to the recommendations of Parnin and Orso [195] who suggested that absolute metrics reflect the actual amount of effort required from developers better than percentages. The accuracy (*acc@n*) calculates the number of cases where the flaky classes were ranked in the top n. In our study, we report the *acc@n* with 1, 3, 5, and 10 as n values. In the cases of multiple flaky classes, we consider the flaky class to be among the top n, if at least one of the flaky classes is. The second metric, wasted effort (*wef*), allows us to measure the effort wasted while searching for the flaky class. It is formally defined as [183]:

$$wef = |susp(x) > susp(x*)| + |susp(x) = susp(x*)|/2 + 1/2$$

Where *susp*() provides the suspiciousness score of the class x, $x*$ is the flaky class, and $|.|$ provides the number of elements in the set. Accordingly, *wef* measures the absolute number of classes inspected before reaching the real flaky class $x*$.

For our approach to be useful for developers, it should provide guidance beyond currently available information. When a program fails due to flaky tests, one thing that can be helpful to identify the cause is a list of classes covered by the flaky tests. Hence, in this paper, we count the total number of classes covered by flaky tests (*i.e.* our baseline) and compare it with the number of classes inspected to locate a flaky class (*i.e.* *wef*+1). More specifically, in addition to the two absolute metrics, we measure the relative effort defined as:

$$R_{wef} = \frac{100 \times (wef + 1)}{\#\text{ of classes covered by flaky tests}},\ 0 < R_{wef} \leq 100$$

If R_{wef} is smaller than 50, we consider our approach to outperform the baseline since it saves more than the expected effort (*i.e.* average) of the baseline.

7.3.6 Tie-breaking

Both SBFL and our evolved formulæ compute an ordinal score for each class. As a result, multiple classes can have the same score, being tied to each other. Ties are generally harmful as they force developers to inspect more classes. Among various tie-breakers introduced and adopted to handle this problem [196], we use a max tie-breaker that assigns the lowest rank (*i.e.* the maximum) to all tied entities. We choose the max tie-breaker to avoid overinterpretation of the results.

7.4 Results

7.4.1 RQ1: Effectiveness

Table 7.4 shows the localisation results of SBFL formulæ. Among the four SBFL formulæ, Dstar yields the worst results both in accuracy and wasted effort, while the other three perform similarly. Out of 38 analysed flaky classes, Dstar ranks 18 (47%) in the top 10. Ochiai, which performs the best, places 53% of flaky classes (*i.e.* 21) within the top 10 and 16% (6) at the top. Nevertheless, regardless of which formula we use, our SBFL-based approach outperforms the baseline of inspecting classes covered by flaky tests: for all four SBFL formulæ, R_{wef} is always smaller than 50 in total, especially in its median. It is worth noting that since the total number of classes covered by flaky tests differs in each flaky commit, R_{wef} does not always concur with *wef*. For Ochiai, R_{wef} reduces to 6, meaning we only need to inspect 6% of the classes covered by flaky tests.

Table 7.5 presents the evaluation results of our GP model evolved to combine the four SBFL formulæ. As explained in Section 7.3.1, we report only the results of the model with the median fitness among 30 models. In contrast to what we expected from combining the four SBFL formulæ using GP, we fail to observe any meaningful improvement compared to the results of Ochiai, the best of the four

Table 7.4: RQ1: Effectiveness of SBFL formulæ. (#) denotes the total number of flaky commits for each project. The row *Perc* contains the percentage of flaky commits whose triggering flaky classes are ranked in the top n; these values are computed only for *acc@n*.

Proj. (#)	**Dstar**						**Ochiai**						**Tarantula**						**Barinel**					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	0	3	4	4	33.0 (17)	7 (5)	2	5	5	5	14.9 (13)	1 (4)	1	4	4	5	11.9 (12)	4 (4)	1	4	4	5	11.6 (12)	4 (4)
ignite (14)	0	2	2	2	214.7 (21)	31 (4)	0	3	3	4	212.0 (20)	20 (4)	0	3	3	4	177.1 (17)	20 (4)	0	3	3	4	175.1 (17)	20 (4)
Pulsar (10)	1	3	6	9	9.9 (21)	4 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)
Alluxio (3)	0	0	0	1	60.7 (43)	72 (31)	0	0	0	1	71.0 (46)	72 (41)	0	0	0	0	92.7 (59)	73 (58)	0	0	0	0	105.3 (66)	87 (65)
Neo4j (3)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	23.0 (43)	1 (18)	1	2	2	2	23.7 (43)	1 (18)
Total (38)	2	10	14	18	94.4 (24)	**11 (17)**	6	15	16	21	90.2 (21)	7 (6)	5	14	15	20	79.3 (21)	8 (7)	5	14	15	20	79.6 (21)	8 (7)
Perc (%)	5	26	37	**47**	-	-	**16**	**39**	42	**55**	-	-	13	37	39	**53**	-	-	13	37	39	**53**	-	-

Table 7.5: RQ1: The effectiveness of GP evolved formulæ using Ochiai, Barinel, Tarantula, and DStar.

Project	**Total**	**acc**				**wef (R_{wef})**	
		@1	@3	@5	@10	mean	med
Hbase	8	1	4	5	5	13.12 (16)	2.5 (5)
Ignite	14	0	3	3	5	214.93 (21)	20.0 (4)
Pulsar	10	3	5	6	9	9.20 (23)	3.0 (9)
Alluxio	3	0	0	0	1	101.67 (65)	86.0 (83)
Neo4j	3	1	2	2	2	23.33 (43)	1.0 (18)
Total	38	5	14	16	22	94.24 (26)	6.5 (8)
Percentage (%)	100	**13**	**37**	42	**58**	-	-

formulæ: the *acc@10* and the median wasted effort improve only marginally, and R_{wef} degrades.

To understand these observations, we inspect the intersection between the sets of classes ranked in the top 5 by these four SBFL formulæ. Figure 7.1 presents this intersection in a Venn diagram. Out of 14,16,15,15 flaky classes ranked within the top 5 by Dstar, Ochiai, Tarantula, and Barinel, 13 of them are the same flaky classes. There are two additional classes that are ranked in the top 5 by all except Dstar and one extra class by only Ochiai and Dstar. Overall, the diagram demonstrates that there are large overlaps between the results of these four SBFL formulæ. Thus, we can conclude that the GP-evolved formula did not lead to substantial improvements because there was no space for improvement as all four input formulæ provided similar signals. This conclusion brings out the need for introducing external signals from other code and change metrics, which will be discussed in the following research question.

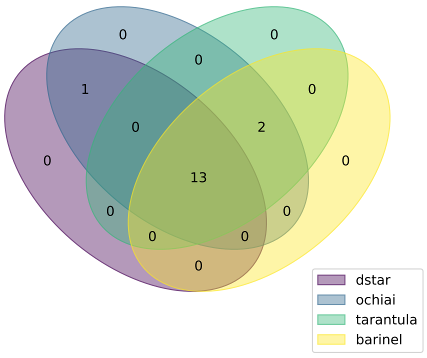


Figure 7.1: Venn-diagram of flaky classes ranked in the top 5 by the four SBFL formulæ.

RQ1: Using SBFL, we were able to localise flaky classes by inspecting only 21-24% (6-7%) of classes covered by flaky tests on average (median). With Ochiai, flaky classes are ranked at the top and in the top 10 for 16% and 55% of total flaky commits.

7.4.2 RQ2: Code and Change Metrics

Table 7.6 shows the evaluation results for GP-evolved models using SBFL scores with change and code metrics. The table shows that the addition of signals from change and size metrics leads to an improvement in the identification of flaky classes. In particular, by adding change metrics, the percentage of classes ranked at the top reaches 24%. This percentage is much higher than the maximum percentage achieved with SBFL alone, which is 16% with Ochiai. On the contrary, we do not observe any significant improvements in the number of flaky tests ranked in the top 10 or top 5. Combined, these results imply that these change and size metrics can give additional signals that break ties between the classes located near the top, allowing developers to identify the exact cause of flakiness more precisely. The comparison with the results of GP with only SBFL formulæ in Table 7.5 further supports the usefulness of change and size metrics. Specifically, by adding change and size metrics, the percentage of flaky classes ranked at the top (*acc*@1) goes from 13% to 24% and 18%, respectively. In addition, average R_{wef} improves 5% with change metrics and 4% with size metrics.

With regard to flakiness metrics, their combination with SBFL scores does not lead to any notable improvements in the ranking of classes at the top. The percentage of classes at the top is 11% and the percentage of classes in the top 10 is 53%. One possible explanation for this is that our flakiness metrics are derived from a flakiness taxonomy that focuses on the test instead of the CUT. Hence, using metrics derived from such categories may not be helpful in the identification of CUT components that are responsible for flakiness. To alleviate this, future studies should consider categories and metrics that are derived from the CUT, and existing flakiness taxonomies should be updated accordingly.

Table 7.6: RQ2: The contribution of flakiness, change, and size metrics to the identification of flaky classes.

Proj. (#)	**SBFL & flakiness**						**SBFL & change**						**SBFL & size**					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	1	4	5	5	11.9 (12)	3 (4)	2	4	4	5	16.9 (13)	4 (4)	2	4	5	5	11.4 (12)	3 (3)
Ignite (14)	0	2	2	4	230.9 (26)	63 (4)	2	4	4	4	222.3 (24)	18 (4)	1	3	3	5	220.1 (24)	43 (4)
Pulsar (10)	2	5	6	8	10.2 (15)	3 (8)	3	5	7	9	8.0 (12)	2 (5)	2	5	7	9	6.9 (13)	2 (6)
Alluxio (3)	0	0	1	1	97.7 (51)	73 (65)	0	0	1	1	75.7 (49)	94 (39)	0	0	1	1	90.7 (49)	77 (58)
Neo4j (3)	1	2	2	2	19.3 (42)	1 (18)	2	2	2	2	6.7 (37)	0 (9)	2	2	2	2	23.0 (40)	0 (10)
Total (38)	4	13	16	20	99.5 (24)	8 (8)	9	15	18	21	94.1 (21)	5 (6)	7	14	18	22	94.3 (22)	5 (7)
Percentage (%)	**11**	34	42	**53**	-	-	**24**	39	47	55	-	-	**18**	37	47	58	-	-

To further investigate the impact of change and size metrics on the identification performance, we analyse the involvement of each metric in our GP-evolved formulæ. Table 7.7 shows the frequency of change and size metrics in the GP evolved formulæ generated under the configuration of using SBFL and change metrics (*i.e.* SBFL & Change) and the configuration of using SBFL and size metrics (*i.e.* SBFL & Size). As shown in this table, both change and size metrics are frequently involved in the final formulæ, confirming that our observed improvement did not come only from using GP. Based on these results, we posit that change and size metrics can contribute positively to the identification of flaky classes.

Table 7.7: Frequency of metrics in GP-evolved formulæ (from 0 to 1). 'Changes' and 'Dev' denote *'Unique Changes'* and *'Developers'*, respectively. The column 'SBFL' contains the average frequency of the four SBFL metrics.

	SBFL	**Changes**	**Dev**	**Age**	**LOC**	**DOI**	**CC**
SBFL & Change	0.45	0.50	0.37	0.53	-	-	-
SBFL & Size	0.50	-	-	-	0.71	0.37	0.73

RQ2: The augmentation of Spectrum-Based Fault Localisation with change or size metrics lets more flaky classes be ranked near the top; by adding change metrics, we can rank 24% flaky classes at the top. In contrast, metrics specific to flakiness categories do not provide any beneficial signals to the identification approach.

7.4.3 RQ3: Ensemble Method

Table 7.8 presents the evaluation results for the voting method with 60 GP-evolved models, half from using SBFL and change metrics and the other half from using SBFL and size metrics. We decided to exclude the models that build on flakiness metrics since their usage did not improve the performance any further. As explained in Section 7.3.3, there can be a case where none of the participating models succeeds to vote for the true candidate since individual models vote only for those ranked within the top n. For this case, we report the median of all rankings of the models as an alternative.

The results show that the voting step further improves the ranking results. The most notable improvement is the accuracy at the top 3, which reaches 47%. Although the improvements in the other accuracy metrics are not as noticeable as what we have seen in the accuracy at the top 3, there are constant improvements over the results without voting. The average of wasted effort remains almost the same while the median improves from the voting, dropping to 3.5. These results imply that the voting allows those near the top to shift further to higher ranks based on the agreement among the models that exploit and capture different features of flaky classes. Nonetheless, the constant improvements in R_{we}, both per project and in total, suggest that through the voting, we can rank flaky classes further near to the top; for example, in Alluxio, where R_{wef} is always near 50, average R_{wef} reduces to 22 and its median to 10. These results imply that voting can leverage the complementarity between different models, further improving the localisation of flakiness.

Table 7.8: RQ3: The effectiveness of the voting between 60 different GP-evolved models, 30 from SBFL with change metrics, and 30 from using SBFL with size metrics. 'Perc' denotes Percentage

Project	**Total**	**acc**				**wef ($\mathbf{R}_{wef}$)**	
		@1	@3	@5	@10	mean	med
Hbase	8	3	5	6	6	9.62 (12)	1.5 (2)
Ignite	14	2	4	4	4	228.61 (24)	17.5 (4)
Pulsar	10	3	6	7	9	7.30 (12)	2.0 (5)
Alluxio	3	1	1	1	2	61.83 (22)	9.0 (10)
Neo4j	3	1	2	2	2	19.67 (42)	1.0 (18)
Total	38	10	18	20	23	94.61 (19)	3.5 (5)
Perc (%)	100	26	**47**	53	61	-	-

RQ3: A voting between models based on SBFL, change, and size metrics, provides the best ranking for flaky classes. 47% of flaky classes are ranked in the top 3 and 26% of them are ranked at the top. The average R_{wef} further reduces to 19, highlighting the practical usefulness of our approach.

7.4.4 RQ4: Flakiness Categories

Table 7.9 presents the performances of the voting method on the different flakiness categories encountered in our dataset. The *"Ambiguous"* category represents cases where the flaky tests could not be assigned to any of the known flakiness categories. First, we observe that the most common categories are Concurrency and Asynchronous Waits. This aligns with observations from previous studies [49], [58], [62] and confirms that the taxonomy adopted for our metrics is adequate for our distribution. Furthermore, we observe a discrepancy between the performances in different categories. Classes responsible for Async Waits are well identified with 80% of the classes in the top 10, and 30% of them at the top. Classes responsible for Concurrency also show good performances with 50% of them in the top 10, and 38% of them at the top; the average R_{wef} is below ten, eight precisely, meaning we can locate flaky classes by inspecting less than 10% of the total number of the classes covered by flaky tests.

Categories such as Time and I/O show much lower performances, with 33% and 0% of flaky classes in the top 10, respectively. Nevertheless, given the low number of instances for these categories, it is hard to discuss or generalise their results. With only two instances, the category Unordered Collections shows curious results as one class is ranked second and the other one ranked 663. To understand the reasons behind the bad ranking, we manually inspected this case[3]. We found that the concerned test, `testUnstableTopology`, was executed twice due to a retry mechanism. Both executions led to failure, but interestingly, we found that the two failures have different causes. One of them is due to a lack of context initialisation and is likely to be the reason behind flakiness. As the two failure causes are different, the coverage is also different in them. Specifically, one of the failures did not cover the flaky class, and as the coverage of this failure was leveraged in the SBFL, the flaky class was not considered suspicious. We discuss other reasons responsible for poor ranking in Section 7.5.

[3]`https://github.com/apache/ignite/commit/188e4d52c2`

[4]One flaky class belongs to two categories: Network and Unordered Collections.

Table 7.9: RQ4: The effectiveness per flakiness category

Flakiness Category	**acc** @1	@3	@5	@10	**wef ($\mathbf{R}_{wef}$)** mean	med
Concurrency (16)	6 (**38**)	7 (44)	7(44)	8 (**50**)	147.53 (27)	9.5 (9)
Async wait (10)	3 (**30**)	6 (60)	8 (80)	8 (**80**)	21.05 (8)	1.5 (3)
Ambiguous (4)	1 (25)	2 (50)	2 (50)	3 (75)	18.88 (5)	3.5 (5)
Time (3)	0 (0)	0 (0)	0 (0)	1 (**33**)	88.33 (16)	14.0 (10)
Network (2)	0 (0)	2 (100)	2 (100)	2 (100)	1.00 (10)	1.0 (10)
Unordered collections (2)	0 (0)	1 (50)	1(50)	1 (50)	331.5 (33)	331.5 (33)
I/O (1)	0 (0)	0 (0)	0(0)	0 (**0**)	12.50 (3)	12.5 (3)
Random (1)	0 (0)	1 (100)	1 (100)	1 (100)	2.00 (75)	2.0 (75)
Total (39[4])	10	18	20	23	94.47 (19)	3.5 (5)
Perc (%)	26	47	53	61	-	-

RQ4: The most prominent flakiness categories, Concurrency and Asynchronous Waits, are identified effectively, with 38% and 30% of their flaky classes ranked at the top, respectively. In the Concurrency category, flaky classes are identified by examining 8% of classes covered by flaky tests on average.

7.5 Discussion

In this section, we discuss our results in light of the existing literature on test flakiness and fault localisation. Our approach uses existing fault localisation techniques to identify flaky classes in the CUT. While we leverage various data sources, the main strength of our approach comes from adopting existing SBFL techniques, as explained in RQ2. The effectiveness of other data, such as change metrics, is limited in providing additional signals that break ties between the classes already ranked near the top. Hence, the performance of our approach largely depends on the applicability of SBFL to our flaky class identification problem.

The flaky class identification problem and traditional fault localisation problems are similar in the way they are debugged (*i.e.* from the reproduction and cause identification to the fix). As described in 7.3.1, this resemblance allows us to redefine SBFL techniques to identify flaky classes instead of faulty ones. Nevertheless, there is one significant difference between them: the characteristics of a test suite.

Many fault localisation studies assume a test to cover a single functionality, and the subjects they studied often follow this assumption [174], [175], [197]. In contrast, we did not consider such an assumption for test subject selection to reflect a realistic scenario of flaky test failure. This difference may restrict the applicability of existing fault localisation techniques to the flaky class identification problem, especially test coverage-based techniques, such as SBFL. Indeed, although we identified 26% and 61% of flaky classes at the top and within the top 10, we failed to reach the performance reported in prior work on fault localisation [180]. Hence, we investigate the diagnosability of the test suite of our subjects using the Density, Diversity, and Uniqueness (DDU) metric [198].

DDU diagnoses the adequacy of SBFL for a software system by considering three properties of its test suite: Density, Diversity, and Uniqueness. Each property covers a distinct feature of a test suite, and DDU is computed as the multiplication of these three properties. Density evaluates how frequently a code entity, in our case a class, is covered by tests. Diversity is about whether tests cover code entities in a diverse fashion. Lastly, uniqueness guarantees that different code entities are covered by different sets of tests. All these three components of DDU have values between 0 and 1. The higher the DDU is, the more adequate the test suite is for SBFL.

Table 7.10 presents DDU values for the five projects analysed in this study. While all five projects generally have high diversity values (*i.e.* all above 0.9), they have relatively low uniqueness and density values, which results in low DDU scores. Among the five projects, Pulsar has the highest DDU score of 0.289, followed by Neo4j, Alluxio, Ignite, and Hbase. Since both Neo4j and Alluxio have only three flaky classes, which might be too small to discuss the identification results, we will skip these two for the following discussion. Among the remaining three projects, all our flaky class identification methods, ranging from pure SBFL to voting, perform the best on Pulsar, the one with the highest DDU score, in *acc@n* and *wef*. For instance, even the pure SBFL approach that often performs the worst successfully localised nine out of ten flaky classes of Pulsar within the top 10 and more than half within the top five. The same trend was observed in both GP and voting-based methods. Compared to HBase, while Ignite has a slightly higher average for the DDU score, it has a far lower Uniqueness score (*i.e.* 0.188 for Ignite and 0.413 for HBase). Uniqueness evaluates whether a code entity is distinguishable; we assume that the flaky classes have different coverage than non-flaky classes. Thus, we suspect that Ignite having a lower Uniqueness is why our methods were not as effective on Ignite as on HBase: we have the worst results on Ignite in both absolute (*i.e.* *acc@n* and *wef*) and relative effort (*i.e.* R_{wef}).

Based on these results, we argue that while our outcome may not be as good as those reported by prior fault localisation studies[186], [187], that is mainly due to the inherently low diagnosability of a test suite (*e.g.* covering too many classes in the same fashion). This test-suite adequacy issue commonly exists in the fault localisation field[193] and is not limited to flaky class identification. Hence, we posit that the performance of our approach can improve along with the advances in fault localisation techniques.

Table 7.10: DDU metrics for the analysed test suites.

Project	**Density**			**Diversity**			**Uniqueness**			**DDU**		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Hbase	0.049	0.477	0.248	0.995	0.999	0.997	0.188	0.553	0.413	0.021	0.116	0.091
Ignite	0.368	0.993	0.736	0.918	1.000	0.979	0.045	0.486	0.188	0.034	0.466	0.132
Pulsar	0.029	0.998	0.491	0.984	0.998	0.994	0.520	0.786	0.609	0.019	0.518	0.289
Alluxio	0.414	0.833	0.591	0.958	0.996	0.982	0.226	0.615	0.362	0.101	0.322	0.201
Neo4j	0.127	0.739	0.515	0.894	0.993	0.931	0.268	0.791	0.585	0.088	0.522	0.258

In an attempt to shed light on the 15 cases where the class was ranked outside the top 10 by our voting approach, we extended our inspection to reason about such performances. We observed that flaky tests covering a high number of classes are more likely to result in low performances. For example, the flaky test `shutdownDatabaseDuringIndexPopulations` in Neo4j covers 480 classes and its flaky class was ranked 59 by our voting approach whereas the other flaky tests in Neo4j (having their corresponding flaky classes ranked 1 and 2) cover fewer than 10 classes. When we inspect the DDU score of the specific commit that contains this test, it has a relatively low DDU score compared to the other two commits. Additionally, most of the mis-ranked classes are found in the Ignite project (10/15), whose DDU score is the second-lowest, and its tests cover on average 492 classes. Due to this consequent number of covered classes, we suspect these tests to be of a higher level, *i.e.* integration or end-to-end tests. This aligns with studies highlighting the prevalence of flakiness in integration and system tests [103], [104]. Still, our approach does not systematically fail to identify flaky classes covered by higher-level tests as nine of them (flaky test covering more than 100 classes) are listed in the top 10.

7.6 Threats to Validity

7.6.1 External Validity

The main threat to the external validity of this study is the dataset size. To ensure the generalisability of our results, it would have been preferable to include more flaky tests in our experiments. Nonetheless, the datasets of flaky tests are generally limited in size due to the elusiveness of flakiness [94], [96], [199]. Moreover, as explained in Section 7.2, the requirements of this study limited the set of candidates considerably. For a commit to be eligible in our study, it needs to have atomic changes fixing flakiness in the CUT. However, only 24% of flaky tests actually stem from the CUT, which limits the size of potential subjects [58]. Besides, the creation of our dataset required a substantial amount of manual work to identify suitable commits and perform necessary changes to retrieve coverage matrices. For instance, for each commit, we had to modify the build script to match GZOLTAR requirements, *i.e.* find the test executor version that matches both the program under test and the plugin. We iteratively removed non-essential listeners and other plugins that could interfere with the instrumentation. Moreover, we had to find and adapt the execution environment to match the program under test and the testing environment. Finally, compared to the works of Lam *et al.* [85] and Zitfci and Cavalcanti [86], which were conducted on proprietary software, this study is the first to leverage open-source software to localise flakiness root causes. Thus, our dataset and ground truth can be valuable for future studies on flakiness debugging.

7.6.2 Internal Validity

One potential threat to our internal validity is our definition of flakiness root causes within the CUT, *i.e.* flaky classes. We rely on flakiness-fixing commits to identify classes that are responsible for flakiness. However, we cannot be certain that (i) the flakiness fix is effective, and (ii) the modified class is the one responsible for flakiness. Indeed, a study by Lam *et al.* [62] showed that developers may wrongly claim that their commits fix flaky tests before realising that the fix is ineffective. Additionally, there are no guarantees that the classes included in the fix are the ones responsible for flakiness. Nonetheless, if the class was part of the proclaimed fix, this means that the developers found it, at least, relevant. Hence, its identification by our approach is still helpful for developers willing to understand, debug, and fix flaky tests.

7.6.3 Construct Validity

One potential threat to our construct validity is our measurement of the coverage for flaky tests. A flaky test can pass and fail for the same version of the program, but in practice, it may be extremely difficult to reproduce both the pass and failure [96], [137]. Hence, a test can be observed as flaky by the project developers and therefore fixed, yet we are unable to reproduce the pass and failure in our experiments even with a large number of reruns [96]. For this reason, we focused on the available status, *i.e.* pass or failure, and retrieve its coverage. It is possible that including the coverage of both the pass and failure from the flaky tests might lead to different results with spectrum-based fault localisation. Thus, we encourage future studies to investigate this direction. Another possible threat is whether the evaluation results of our approach truly support what we claim. We use two absolute metrics, $acc@n$ and wef, that can reflect the realistic debugging effort of developers, following the suggestion from Parnin and Orso [195], and one relative metric, R_{wef}, to compare with the baseline of inspecting classes covered by flaky tests.

7.7 Conclusion

Root-causing flaky tests is a young nonetheless important research area. Surveys showed how difficult flakiness debugging can be for developers, even when knowing which test is flaky (post-detection) [49], [105]. We leveraged our results to elaborate on a few recommendations for future works. We presented the first empirical evaluation of SBFL as a potential approach for identifying flaky classes. We investigated three approaches: pure SBFL, SBFL augmented with change and code metrics, and an ensemble of them. We evaluated these approaches on five open-source Java projects. Our results show that SBFL-based approaches can identify flaky classes relatively well, especially with code and change metrics, suggesting that code components responsible for flakiness exhibit similar properties with faults. This finding highlights the potential of existing fault localisation techniques for flakiness identification. At the same time, the results show that flaky tests can have unique failure causes that may mislead any coverage-based root cause analysis, stressing the need to consider these flakiness-specific causes in future studies.

Our study forms the first step towards flakiness localisation. We believe that there is a lot of room for improvement and encourage future studies to explore additional techniques, fault prediction metrics, and devise techniques that can further improve and support flakiness localisation.

8 The Importance of Discerning Flaky from Fault-triggering Test Failures

While promising, the actual utility of the methods predicting flaky tests remains unclear since they have not been evaluated within a continuous integration (CI) process. In particular, it remains unclear what is the impact of missed faults, i.e. the consideration of fault-triggering test failures as flaky, at different CI cycles. In this chapter, we apply state-of-the-art flakiness prediction methods at the Chromium CI and check their performance. Perhaps surprisingly, we find that the application of such methods leads to numerous faults missed, which is approximately $^3/_4$ of all regression faults. To explain this result, we analyse the fault-triggering failures and find that flaky tests have a strong fault-revealing capability. Overall, our findings suggest that future research should focus on predicting flaky test failures instead of flaky tests (to reduce missed faults) and reveal the need for adopting more thorough experimental methodologies when evaluating flakiness prediction methods (to better reflect the actual practice).

This chapter is based on the work in the following paper:

- G. Haben, S. Habchi, M. Papadakis, *et al.*, *The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci*, 2023. arXiv: `2302.10594 [cs.SE]`

Contents

8.1 Introduction

Continuous Integration (CI) is a software engineering process that allows developers to frequently merge their changes in a shared repository [138]. To ensure a fast and efficient collaboration, the CI automates parts, if not all, of the development life cycle. Regression testing is an important aspect of CI as it ensures that new changes do not break existing functionality. Test suites are executed for every commit and test results signal whether changes should be integrated into the operational codebase or not.

Tests are an essential part of the CI as they prevent faults from enterring the codebase, and they ensure smooth code integration and overall good software function. Unfortunately, some tests, named flaky tests, exhibit a non-deterministic behaviour as they both pass and fail for the same version of the codebase. When flaky tests fail, they send false alerts to developers about the state of their applications and the integration of their changes.

Indeed, developers spend time and effort investigating flaky failures, as they can be difficult to reproduce, only to discover that they are false alerts [49]. These false alerts occur frequently in open-source and industrial projects [57], [59], [83], [103] and make developers lose not only time but also their trust in the test signal. This trust issue in turn introduces the risk of ignoring fault-triggering test failures. This way, false alerts defy the purpose of software testing and hinder the flow of the CI.

To deal with test flakiness, many techniques aiming at detecting flaky tests have been introduced. A basic approach is to rerun tests multiple times and observe their outcomes. While to some extent effective, test reruns are extremely expensive [57], [83] and unsafe. To this end, researchers have proposed several approaches relying on static (the test code) [57], [91], [92], [97], [152] or dynamic (test executions) [59], [83], [86] information (or both) [96] to predict whether a given test is flaky.

Among the many flakiness prediction methods, the vocabulary-based ones [92]–[95], [101] are the most popular [74]. They rely on machine learning models that predict test flakiness based on the occurrences of source code tokens of the candidate tests. Interestingly, previous research has found these approaches particularly precise, with current state-of-the-art achieving accuracy values higher than 95% [92], [94], [95], [97], [152].

At the same time, vocabulary-based approaches are static and text-based, thus, they are both portable, *i.e.* limited to a specific language, and interpretable, *i.e.* users may understand the cause of flakiness based on the keywords that impact the model's decisions. All these characteristics (precision, portability and interpretability) make vocabulary-based approaches appealing; they are flexible and easy to use in practice. In view of this, we decided to replicate these techniques on an industrial project (the Chromium project) and evaluated their ability to effectively support the detection of flaky tests during the CI operation cycles.

Perhaps not surprisingly, we found a similar prediction performance (99.2% precision 98.4% recall) to the ones reported by previous studies. Surprisingly though, we noticed numerous fault-triggering failures (approximately 76% of all fault-triggering failures) being marked as flaky by these prediction methods. This means that, in the case where the Chromium teams were to follow the recommendations of these techniques, they would have missed at least 76% of all regression faults (considering their respective test failures as flaky) that were captured by their test suites.

The significantly high fault loss experienced (fault-triggering failures considered as flaky) motivated the investigation of the fault-triggering failures. To this end, we made the following three findings:

- *Flaky tests have a strong fault-revealing capability*, *i.e.* more than 1/3 of all regression faults are triggered by a test that exhibits flaky behaviour at some point in time. This means that methods aiming at detecting flaky tests, inevitably flag as flaky fault-triggering test failures made by these tests. This indicates an inherent limitation of all methods focusing on identifying flaky tests, a fact that is largely ignored by previous studies.
- *Many fault-revealing tests have characteristics similar to flaky ones* and thus, are mistakenly flagged as flaky by the flakiness prediction methods. In our data, the set of fault-triggering failures made by non-flaky tests that are mistakenly predicted as flaky represents 56.2% of all fault-triggering failures.
- *The majority of the flaky tests fail frequently (87.9% of the flaky tests have also flaked in the past)*, making prediction methods mark them as flaky based on the test history, instead of the characteristics of their failures. In particular, a dummy method that classifies as flaky any test that flaked at least once in the past achieves precision and recall values of 99.8% and 87.8% when predicting flaky test failures.

The above findings motivate the need for techniques focusing on flaky test failures, instead of flaky tests, *i.e.* discriminating between fault-triggering and flaky test failures, an essential problem that has largely been ignored by previous research [74]. Therefore, we adapt the vocabulary-based methods for failure-focused predictions and check their performance. Although they miss fewer fault-triggering test failures than the test-focused methods (20.3% FPR compared to 76.2% FPR), their ability to detect fault-triggering failures remains poor, MCC value of 0.25.

With the hope of improving the methods' performance, we augment their feature set with dynamic features related to (flaky) test executions (*e.g.* run duration and historical flake rate) that we find useful for flakiness diagnosis. We achieve a better –but still not acceptable performance – MCC value of 0.42, indicating an improvement to detect fault-triggering failures when considering dynamic features.

Overall, our study demonstrates the need for methods that can effectively predict flaky test failures (instead of flaky tests), to reduce the faults missed, and the need for adopting more thorough experimental methodologies, reflecting the needs of the domain of the actual practice (not just classification metrics), when evaluating flakiness predictions.

In summary, the contributions of our paper are:

- We present *a large empirical study on flakiness prediction, based on the Chromium project* – one of the biggest open-source industrial projects – involving 10,000 builds, more than 200,000 unique tests and 1.8 million test failures. Our study is the first to study the suitability of applying flakiness prediction into a CI pipeline by focusing on the potential losses that they introduce: missed fault-triggering failures.
- We provide empirical evidence that flaky test prediction methods, despite being very precise, are practically non-actionable since they *flag as flaky a majority, approximately 76.2%, of all fault-triggering failures* (56.2% due to misclassifications of fault-revealing tests and 20% due to correct classification of flaky tests that reveal faults).
- We provide empirical evidence that *flaky tests have strong fault-revealing capabilities*, indicating an inherent limitation of existing methods. At the same time, our results motivate the need for failure-focused prediction techniques. Unfortunately, we also show that existing vocabulary-based methods are insufficiently precise, calling for additional research in this area.
- We investigate whether *imbuing the training data with additional dynamic features can enhance the failure prediction effectiveness.* These results show improvement (MCC values are up to 0.42), but indicate that more work remains to be done to develop deployable flakiness prediction for real-world systems.

To support future research, we share our dataset, experimental data and related code, in a replication package.[1]

8.2 Chromium

8.2.1 Overview

Started in 2008, with more than 2,000 contributors and 25 million lines of code, the Chromium web browser is one of the biggest open-source projects currently existing. Google is one of the main maintainers, but other companies and contributors are also taking part in its development.

Chromium relies on *LuCI* as a CI platform [201]. It uses more than 900 parallelized builders, each one of them used to build Chromium with different

[1]https://github.com/serval-uni-lu/DiscerningFlakyFailures

settings (*e.g.* different compilers, instrumented versions for memory error detection, fuzzing, etc) and to target different operating systems (*e.g.* Android, Mac OS, Linux, and Windows).

Each builder is responsible for a list of builds triggered by commits made to the project. If a builder is already busy, a scheduler creates a queue of commits waiting to be processed. This means that more than one change can be included in a single build execution if the development pace is faster than what the builders can process. Within a build, we find details about build properties, start and end times, status (*i.e.* pending, success or failure), a listing of the steps and links to the logs.

At the beginning of the project, building and testing were sequential. Builders used to compile the project and zip the results to builders responsible for tests. Testing was taking a lot of time, slowing developers' productivity and testing Chromium on several platforms was not conceivable. A swarming infrastructure was then introduced in order to scale according to the Chromium development team's productivity, to keep getting the test results as fast as possible and independently from the number of tests to run or the number of platforms to test. Currently, a fleet of 14,000 build bots runs tasks in parallel. This setup helps to run tests with low latency and handle hundreds of commits per day [202].

In this study, we focus on testers, *i.e.* builders only responsible for running tests. They do not compile the project: when triggered, they simply extract the build from their corresponding builder and run tests on this version. At the time of writing, we found 47 testers running Chromium test suites on distinct operating systems versions. About 200,000 tests are divided into different test suites, the biggest ones being *blink_web_tests* (testing the rendering engine) and *base_unittests* with more than 60,000 tests each.

For each build performed by any tester, we have access to information about test results. Figure 8.1 illustrates the decision process followed by LuCI to determine a test outcome in a specific build. A test is labelled as *pass* when it successfully passed after one execution. In case of a failure, LuCI automatically reruns the test up to 5 times. If all reruns fail, the test is labelled as *unexpected* and will trigger a build failure. In the remaining, we will be referring to *unexpected* tests as *fault-revealing tests*. If a test passes after having one or more failed executions during the same build, it is labelled as *flaky* and will not prevent the build from passing.

8.2.2 Example of a Flaky Test

Figure 8.2 shows a flaky test found in build 119,039[2] of the Linux Tester. This test, `printing/webgl-oversized-printing.html`, ensures that no crash happens

[2]https://ci.chromium.org/ui/p/chromium/builders/ci/Linux%20Tests/119039/

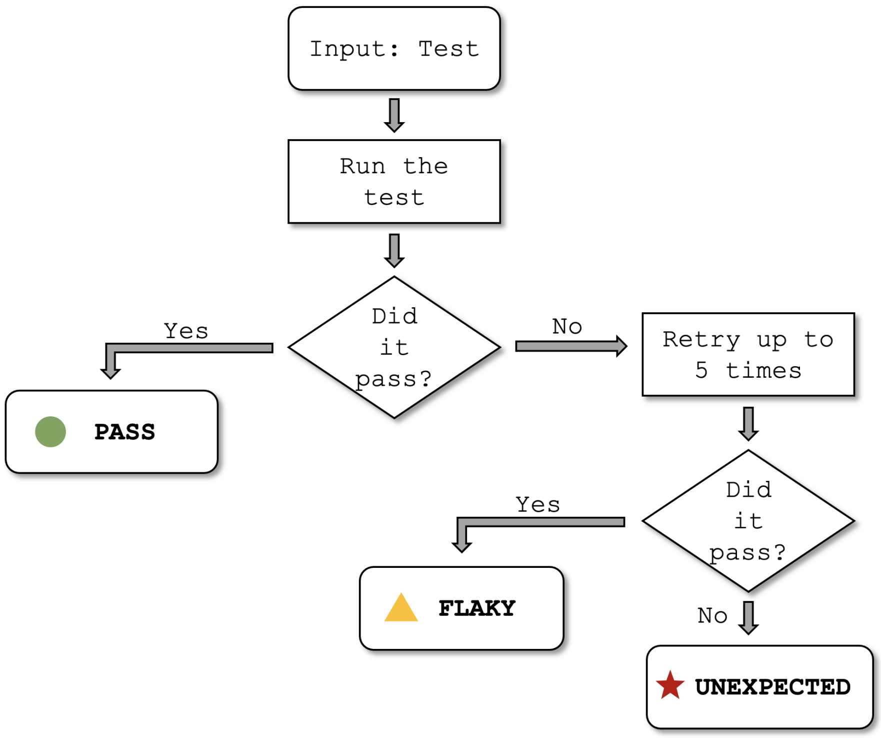


Figure 8.1: Decision tree representing how test outcomes are determined in a build by the Chromium CI. ● PASS depicts successful tests, ▲ FLAKY depicts tests that passed after failing at least once, while ★ UNEXPECTED depicts tests that persistently failed.

on the main thread of the rendering process when using the system. Unfortunately, on its first execution, the test failed after 31 seconds. The run status indicates that a TIMEOUT happened. On the second execution, the test passed after 15 seconds and thus was labelled as flaky. In this case, an issue has been opened in Chromium's bug tracking system.[3] Developers state that *"this test makes a huge memory allocation in the GPU process which intermittently causes OOM and a GPU process crash"*.

TIMEOUT is a run status that intuitively leads to possible flakiness, as we can easily think of other executions of the same test being completed before reaching the time limit. In addition to this feature, we can also look for hints of flakiness in the source code. As with many UI tests in Chromium, this one is handled by `testRunner`: a test harness in charge of their automatic executions. We can see the `testRunner` making a call to `waitUntilDone()` on line 19. Vocabulary about waits is common in Chromium's web tests. This keyword, for example, could

[3] https://bugs.chromium.org/p/chromium/issues/detail?id=1393294

```
 3  <head>
 4  <script src="../resources/js-test.js"></script>
 5  </head>
 6  <body>
 7  <div id="console"></div>
 8  <script>
 9  var can = document.createElement('canvas');
10  can.width = can.height = 7326;
11  document.body.appendChild(can);
12  var ctx = can.getContext("webgl");
13  ctx.clearColor(0, 0, 0, 1);
14  ctx.clear(ctx.COLOR_BUFFER_BIT);
15
16  if (!window.testRunner) {
17      testFailed("Requires window.testRunner");
18  } else {
19      testRunner.waitUntilDone();
20      testRunner.capturePrintingPixelsThen(() => {
21        testPassed("Printed without crashing.");
22        testRunner.notifyDone();
23      });
24  }
25  </script>
26  </body>
```

Figure 8.2: An example of a flaky test caused by a timeout. The test consists of an HTML file `printing/webgl-oversized-printing.html`, build 119,039 of the Linux Tester. The call to WAITUNTILDONE() on line 19 is likely the reason for the failure.

potentially be leveraged by flakiness detectors to classify tests or failures.

8.3 Data

8.3.1 Definitions

Some of our definitions are slightly different from the ones used by previous work since Chromium has its specific continuous integration setup. To make things clear, we define the elements we will discuss in this section. In the scope of *a single build*, we employ the following definitions:

- **Fault-revealing test**: A test that consistently failed after reruns in the same build, revealing a regression fault.
- **Flaky test**: A test that failed once or more and then passed after reruns in the same build.

- **Flaky failure**: A test failure caused by a flaky test.
- **Fault-triggering failure**: A test failure caused by a fault-revealing test.

8.3.2 Data Collection

To perform our study, we collected test execution data from 10,000 consecutive builds completed by the Linux Tester by querying the LuCI API. This represents a period of time of about 9 months taken between March 2022 and December 2022.

Table 8.1 summarises the list of information extracted and computed for all tests executed in all builds. The BUILDID corresponds to the build in which tests were executed. RUNDURATION is the execution time spent to run the test. RUNSTATUS gives information about the run result (*e.g.* passing, failing, and skipped) and RUNTAGSTATUS returns more precise information about the result of a run depending on the type of test or test suite (*e.g.* timeout and failure on exit). We retrieved information about the tests' source code by querying Google Git [4]. As builds often handle several commits, we select the revision corresponding to the head of the blame list: the one on which tests were executed. The TESTSUITE is simply the name of the test suite the test belongs to and TESTID is a unique identifier for a test composed of the test suite and the test name (the same test name can be present in different test suites).

All the scripts used to collect the data alongside the created dataset are available in our replication package.

8.3.3 Computing the Flake Rate

The historical sequence of test results is a valuable piece of information commonly used in software testing at scale [57], [103]. We analyse the history of fault-revealing tests and flaky tests by relying on their flakeRate.

This means that for a test t failing (due to flaky or fault-triggering failure) in a build b_n, we analyse all the builds from a time window w (*i.e.* from b_{n-w} to b_{n-1}) to calculate its flake rate as follows:

$$flakeRate(t, n) = \frac{\sum_{x=n-w}^{n-1} flake(t, x)}{w} \tag{8.1}$$

where $flake(t, x) = 1$ if the test t flaked in the build b_x and 0 otherwise. This metric allows us to understand if the flakiness history of a test can help in the flakiness prediction tasks. The test execution history (*a.k.a.* heartbeat) has been used in multiple studies (especially industrial ones [57], [103]) to detect flaky tests. These studies assume that many flaky tests have distinguishable failure patterns

[4]https://chromium.googlesource.com/chromium/src/+/HEAD/

Table 8.1: Description of our features. Column *Feature Name* specifies the identifiers used in our dataset, while Column *Feature Description* details the features

Feature Name	**Feature Description**
buildId	The build number associated with the test execution
flakeRate	The flake rate of the test over the last 35 builds
runDuration	The time spent for this test execution
runStatus	ABORT FAIL PASS CRASH SKIP
runTagStatus	CRASH PASS FAIL TIMEOUT SUCCESS FAILURE FAILURE_ON_EXIT NOTRUN SKIP UNKNOWN
testSource	The test source code
testSuite	The test suite the test belongs to
testId	The test name

over builds and hence can be detected by observing their history. We check whether this assumption also holds in the case of Chromium.

To illustrate the flake rate differences between flaky and fault-revealing tests, we plot the flake rate for both test categories in Figure 8.3. The flake rate is computed using a window of 35 builds. To set this time window, we checked the number of flaky tests having a $flakeRate() = 0$ for build windows ranging from 0 to 40 builds with a step of 5. We observed a convergence at size 35, meaning that higher numbers of builds do not provide additional information.

In the majority of cases, flaky tests have a history of flakiness: the percentage of

Table 8.2: Data collected from the Chromium CI. We used the *Linux Tests* tester, with *10,000* Builds mined over *nine* months. We extracted Passing, Flaky and Fault-revealing tests and their associated Flaky and Fault-triggering Failures.

Tester	**Nb of Builds**	**Period of Time**		**Number of Tests**			**Number of Failures**	
		From	**To**	**Passing**	**Flaky**	**Fault-revealing**	**Flaky**	**Fault-triggering**
Linux Tests	10,000	Mar 2, 2022	Dec 1, 2022	198,273	23,374	2,343	1,833,831	17,171

flaky tests having a $flakeRate() > 0$ is in fact 87.9%. Furthermore, we see a pike for $flakeRate() == 1$, 9.5% of flaky tests were always flaky in their 35 previous builds. Still, there is a non-negligible amount (45.3%) of fault-revealing tests that were flaky at least once in previous builds considered: with a $flakeRate() > 0$. From these observations, we may suggest that the flakeRate() can be used to detect flakiness. Nevertheless, there is still an important overlap between the history of flaky tests and fault-revealing tests.

8.4 Objectives and Methodologies

8.4.1 Research Questions

We start our analysis by assessing the effectiveness of the existing flakiness prediction methods in our project by considering the critical cases where fault-revealing test failures are flagged as flaky by the prediction methods in various CI cycles. We thus are interested in investigating the methods' performance under realistic settings, i.e., correctly detected and missed flaky and fault-triggering test failures, when trained with past CI data and evaluated on future ones. In contrast to previous work, this analysis introduces a new dimension in the evaluation of flakiness predictions which is the investigation of what we lose when adopting a prediction method (the fault-triggering failures classified as flaky). Therefore, we ask:

RQ1: *How well do flaky test prediction methods discern flaky test failures from fault-triggering ones?*

To establish realistic settings, we train the prediction models using the information available (flaky tests and non-flaky ones) at a given point in time, where we have sufficient historical data to train on. We then evaluate the models in subsequent builds with respect to flaky and fault-triggering test failures. We replicate the vocabulary-based methods since they are popular, easy to implement and quite effective, and aimed at learning to predict flaky tests, as done by previous studies.

After checking the prediction performance in a realistic setting (test failures), we repeat the entire process but now we train on historical test failures instead of tests. We make this adaptation with the hope of improving further our predictions

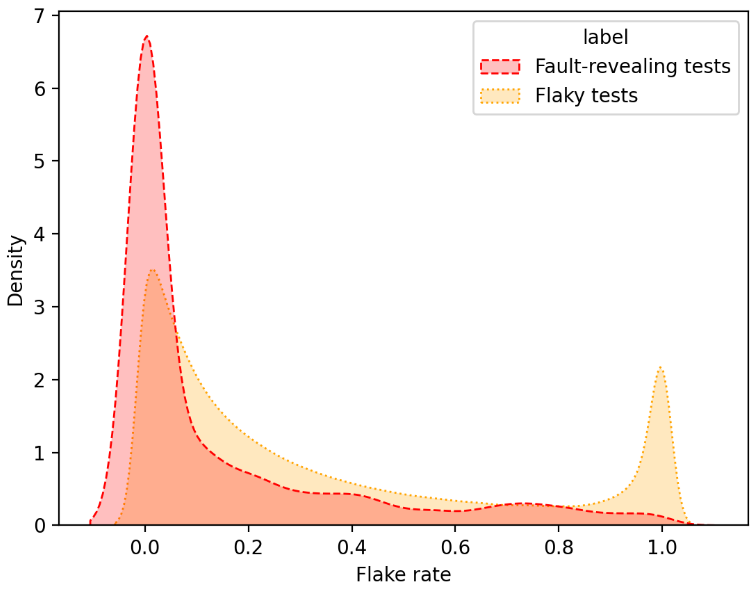


Figure 8.3: Flake rate (*x-axis*) for Flaky and Fault-revealing tests. Density (*y-axis*) is the probability density function. The area under curves integrates to one. Many flaky tests are always flaky in their previous builds. A majority of fault-revealing tests have no history of flakiness at all.

and perhaps improving our understanding of the impact that such predictions may have on missed fault-triggering test failures (those marked by the models as flaky). Hence, we ask:

RQ2: *How well do flaky test failure prediction methods discern flaky test failures from fault-triggering ones?*

Finally, we wish to be comprehensive, so we also optimize and extend the prediction methods with additional features, some of which were suggested by previous studies (the flake rate [103], the run duration [96]) and some dynamic features (test run status, test run tag status, test run duration) that we found by us when experimenting with the flaky tests. Thus, we ask:

RQ3: *Can we improve the accuracy of the flaky test failure predictions by considering dynamic test execution features?*

To answer RQ3, we repeat the analysis carried out for RQ2 but now we are training and optimising for the additional features that we determined during our analysis and check the performance we achieve with regards to test failures, as performed in RQ2.

8.4.2 Experimental Procedure

Selection of a flaky test detection approach

Being a recent topic of interest, several techniques have been introduced in the scientific literature. Approaches relying on code coverage such as FlakeFlagger [96] or DeFlaker [83] are challenging to implement in our case. Chromium's code base consists of several languages and code coverage is both costly and non-trivial to retrieve. Test smells [152] approaches are also difficult to extract as tests are written in many different languages and tools do not always exist.

Having those constraints in mind, we decide to use the vocabulary-based approach introduced by Pinto *et al.* [92]. It received significant attention with several replication studies conducted [94], [95] and follow-up studies and its portability makes it easy to implement regardless of the languages being used.

Training and validation of the existing approaches (RQ1)

We evaluate the ability of the vocabulary-based approach, trained to differentiate flaky from non-flaky tests and used to predict flaky test failures. To do so, we divide our dataset into a training set (containing test information about the first 8,000 builds) and a test set (containing test information about the last 2,000 builds). We train our model following the existing methodologies. The flaky set includes all tests marked as flaky in the training set. The non-flaky set includes all fault-revealing tests and all passing tests in the 8,000th build (*i.e.* the last build of the training set) minus the tests that are found as flaky in any of the builds under study (to increase the confidence of being non-flaky). The test set includes all flaky test failures and all fault-triggering failures (reported by fault-revealing tests). The test set is common in all RQs.

Implementation of a failure classifier (RQ2 and RQ3)

We select flaky failures in our dataset as all failures produced by flaky tests and fault-triggering failures are all failures produced by fault-revealing tests. There are no duplicated data in the case of test failures, as each test execution is unique. For RQ2, we train our classifier on non-flaky executions (passing and fault-revealing tests execution) and flaky failures. In RQ3, we report the performance of a model using execution features (run duration and flake rate).

Time-sensitive evaluation

We split our data in two parts: the first 80% builds are selected as a training set and the last 20% as a holdout set. By doing so, we respect the evolution of failures across time and avoid any data leakage that could occur by randomly selecting data. This time-sensitive aspect is very important to consider. We found that not taking this condition into account and training a model on a shuffled dataset

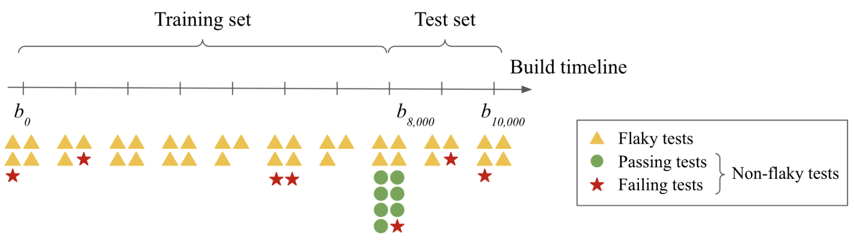


Figure 8.4: The data collected from Chromium's CI consists of flaky, fault-revealing and passing tests spread across 10,000 builds. The build timeline ranges from build b_0 to $b_{10,000}$ and depicts the distribution of the collected tests: flaky tests are spread across all builds and fault-revealing tests happen occasionally. Due to a large number of passing tests, we collected them from the $b_{8,000}$ build (*i.e.* at the end of our training set).

would greatly overestimate the performance. Figure 8.4 shows a representation of our dataset. Flaky tests are present in all builds and Fault-revealing tests are occasional: they happen in 1/4 of builds (See Section 8.6). To mitigate imbalance, we collected all passing tests for 1 build: $b_{8,000}$ and use them in our set of non-flaky tests, for training.

Classifier selection and pipeline description

We use a random forest classifier to perform the predictions. Unfortunately, our dataset is imbalanced with the minority class being 1% of the data. Using a simple random forest would greatly increase the chance of having few or no elements from our minority class in the different trees, making the overall model poor in predicting the class of interest. To alleviate this issue, we decide to use a Balanced Random Forest classifier[203] to facilitate the learning. This implementation artificially modifies the class distribution in each tree so that they are equally represented. Furthermore, we use SMOTE in the training phase to augment data for the minority class [166].

To represent the tests, we use *CountVectorizer* to convert texts as a matrix of token counts. This technique, known as bag-of-words, is used in previous vocabulary-based approaches [92]–[95], [101]. These vectors initially contain as many features as the words appearing in source code of the tests. As the generated dictionary can become big (in terms of size) we need to use feature selection to reduce it, remove irrelevant features (reducing noise in the data) and select the most informative features. Feature selection is thus, helping to reduce the model training time and to improve the overall performance and interpretability of the model.

We use SelectKBest[204] which retains the k highest score features based on the univariate statistical test χ^2. Hyper-parameters of the machine learning pipeline, *i.e.* the number of trees in the forests, the sampling strategy for SMOTE and the number of features to be retained are tuned using a grid search approach and cross-validation in the training set. Once optimized, we retrain a model fitted on the whole training set and evaluate it on the holdout set.

In addition to the precision, recall and MCC metrics, we also report the false positive rate (FPR), that is in our case, the ratio of fault-triggering failures misclassified as flaky over all fault-triggering failures. It is defined by:

$$\mathbf{FPR} = \frac{FP}{FP + TN}$$

8.5 Experimental Results

8.5.1 RQ1: Discerning flaky from fault triggering test failures when training on tests

We trained a model on 69,159 passing tests, 910 fault-revealing tests and 8,857 flaky tests (unique tests). Then, we evaluated it on 217,503 failures caused by flaky tests and 2,320 fault-triggering failures caused by fault-revealing tests. Table 8.3 reports the obtained performance. Similar to the performance achieved by previous vocabulary-based models on other datasets, our model was able to reach high accuracy with a precision of 99.2% and a recall of 98.9%. However, we note a high false-positive rate. This is due to an important amount (76.2%) of fault-triggering failures classified as flaky (FP). This is concerning: fault-triggering failures should not be misclassified as they reflect the existence of real faults. Overall, the MCC value is equal to 0.20, which is relatively low and shows that the model struggles (compared to random selection) to identify fault-triggering failures.

Table 8.3: Vocabulary-based model performance for the prediction of *flaky test failures vs fault-triggering failures* when trained on flaky vs non-flaky (fault-revealing and passing tests). Despite a high accuracy on flaky failures, the low MCC and high FPR show us that it remains challenging for the model to classify negative elements (in our case: fault-triggering failures)

Precision	**Recall**	**MCC**	**FPR**
99.2%	98.9%	0.20	76.2%

The confusion matrix of our model decisions is depicted in Figure 8.5. The x-axis reports the predicted label and the y-axis the actual label. Correct classifications are displayed in the top left (TN) and bottom right (TP). We clearly observe

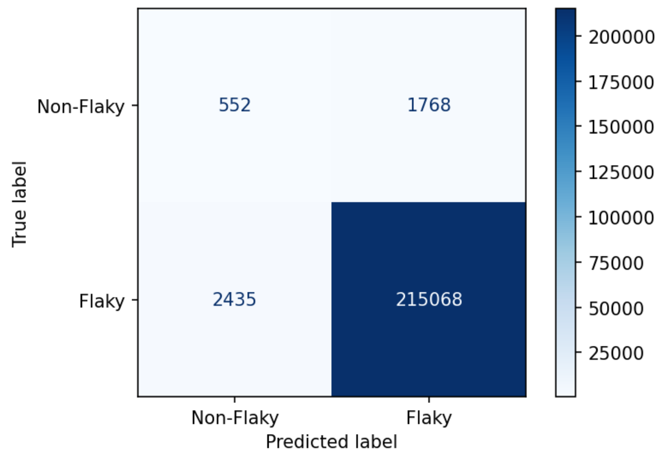


Figure 8.5: Confusion matrix for the vocabulary-based model. High accuracy is reached similar to the performance reported in previous works. Nonetheless, 1,768 (76.2%) out of the 2,320 fault-triggering failures are mislabeled as flaky.

that the model is able to detect flaky tests with high precision. We also see that 2,435 flaky tests are classified as non-flaky (FN). This number is also important to consider: it translates in all cases where developers will be required to investigate irrelevant failures.

We want to further understand the reasons behind the classification of fault-triggering failures. Therefore, we analyse the (fault-revealing) tests causing those failures. Out of the 2,320 fault-triggering failures, 1,768 are in the set of false positives (76.2%) among which we found 463 (20% of all fault-triggering failures) whose tests have a history of flakiness (flakeRate > 0) and 1,305 (56.2% of all fault-triggering failures) without flakiness history. Here it must be noted that depending on the size of the history considered, we may have more tests with past flakiness or less. Overall in our data, 1/3 of all fault-triggering failures are due to tests that have exhibited flakiness behaviour.

RQ1: Similar to previous studies, we report accurate predictions when aiming at flaky tests. However, a high proportion (76.2%) of all fault-triggering failures is classified as flaky (missed faults) and still an important number (2,435) of flaky tests are marked as fault-triggering failures (false alerts).

8.5.2 RQ2: Discerning flaky from fault triggering test failures when training on test failures

The results from RQ1 show that a vocabulary-based model trained to detect flaky tests would still yield an important number of missed faults and false alerts despite having high accuracy. Thus, our goal with RQ2 is to check whether by training our vocabulary-based model we can improve the performance of recognising fault-triggering failures.

Table 8.4 reports the results of such a model. In particular, the first row reports results based on failure training while the second row reports results related to RQ3. Similar to RQ1, we see a high precision and recall, 99.7% and 91.3% respectively, when predicting flaky failures. More interestingly, the MCC slightly increased to 0.25.

RQ2: When training on test failures, solely relying on test code vocabulary as features, to predict if a test failure is flaky or fault-triggering, model performance slightly improves but is still not effective in the context of the Chromium CI.

Table 8.4: Vocabulary-based model performance for the prediction of *flaky failures vs fault-triggering failures* when training on flaky vs non-flaky (fault-triggering and passing test executions). The approach does not work when solely relying on static features (*i.e.* the test source code) and is improved when considering execution features.

Execution features	Precision	Recall	MCC	FPR
No	99.7%	91.3%	0.25	20.3%
Yes	99.5%	98.7%	0.42	42.3%

8.5.3 RQ3: Improving the accuracy of the flaky test failure predictions

In this RQ we check the performance of the vocabulary-based models on the failure classification task when considering additional features from the test executions (run duration, and tests' historical flake rate). These features reflect better the characteristics of the test executions and are linked with test flakiness thereby leading to better results.

In particular, the second row of Table 8.4 reports the related performance. We observe an improvement compared to the model that relies only on vocabulary (RQ2). This new model achieves a similar precision and recall of 99.5% and 98.7% and an improved MCC value 0.42, indicating a better performance in comparison to randomly picked selections. We see that the FPR increased to 42.3%. Together, the results can be explained by fewer false alerts: flaky failures being marked as fault-triggering by the model.

RQ3 When equipped with execution-related features, the vocabulary-based prediction methods do a better job of distinguishing flaky failures from fault-triggering failures (0.42 MCC). Still, the need remains for dedicated methods to successfully learn this challenging classification task.

8.6 Discussion

We seek to better understand the results showing that existing approaches targeting the detection of flaky tests missed a non-negligible part (76.2%) of fault-triggering failures by classifying them as flaky. To do so, we investigate the following aspects regarding *the entire dataset*.

We first report general information about the prevalence of flaky tests and fault-revealing tests in order to have a better view of the failures occurring in each build. Then, we report the number of fault-revealing tests also found as flaky by the Chromium CI (reruns) in other builds (we further refer to them as fault-revealing flaky tests). Finally, we also check for the number of failing builds that only contain fault-revealing flaky tests. We consider these builds important since the related faults are not detected by any non-flaky test and would be missed if flaky test detectors were used.

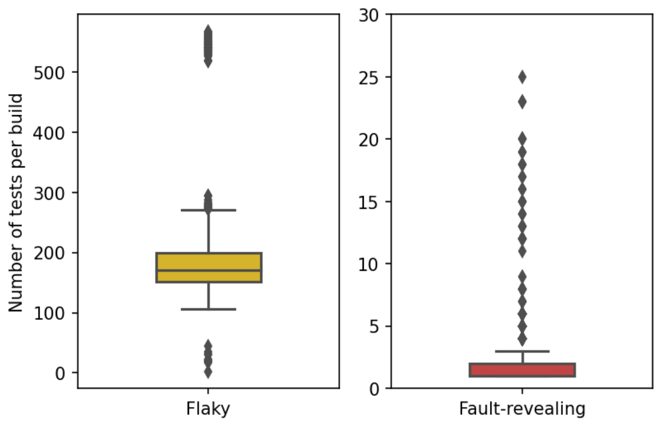


Figure 8.6: Number of flaky tests and fault-revealing tests per build. On average, there are 250 flaky tests per build and 1 fault-revealing test per failing build.

Figure 8.6 shows the distribution of flaky tests and fault-revealing tests in the studied builds. We observe that there is an average of 178 flaky tests per build with a low standard deviation (41), showing that flakiness is prevalent in the Chromium CI. In the case of fault-revealing tests, taking into account all builds would result in an average number of tests close to 0 as a majority of builds are exempt from them. Thus, for better visualisation, we only considered builds containing at least one fault-revealing test (*i.e.* failing builds). The average number of fault-revealing tests per failing build is 2.7.

The standard deviation for fault-revealing tests is 14.9 and the number of fault-revealing tests reported in one build goes up to 579 in our dataset.

Table 8.5: Number of builds containing each studied test type. All builds contain flaky tests. 1/4 contain fault-revealing tests. Among the failing builds, 3/4 contain only fault-revealing tests that are flaky in other builds.

Builds containing	**Number**
Flaky tests	10,000
Fault-revealing tests	2,415
Fault-revealing flaky tests	1,974
Exclusively fault-revealing flaky tests	1,766

Table 8.5 provides, for each type of test, the number of builds that contain at least one instance of this type. We note that all builds contain at least one flaky test (a test that flaked during this build). In Chromium CI, flaky tests are non-blocking and will not cause a build failure. That is, tests flaking within the build are ignored during this build.

Developers are expected to investigate test failures only when they occur consistently across 5 reruns (resulting in a fault-revealing test). Such fault-revealing tests occur in 24.15% of the builds (i.e. in 2,415 builds). Interestingly, 1,974 of these builds (i.e. 81.73%) contain fault-revealing tests that flaked in previous builds, indicating that *tests with a flake history should not be ignored in future builds.* Perhaps worse, in 1,766 builds *all* fault-revealing tests have flaked in some previous builds, indicating that no "reliable" tests identified the fault(s).

By investigating the status of all tests across all builds – see Figure 8.7. Among the 209,530 tests of the Chromium project, 24,820 have failed in at least one build, including 22,477 that were always flaky. Thus, 2,343 tests were fault-revealing in at least one build, i.e., they attested the presence of faults, 897 were also flaky in at least one other build. That is, *38.3% of tests that have been useful to detect faults have also a history of flakiness.*

Flakiness affects all Chromium CI builds and mixes critical (fault-revealing) signals with false (flakiness) signals. Indeed, 81.7% of builds contain fault-revealing tests that were flaky in some previous builds, and 38.3% of all tests flake in some builds and reveal faults in other builds.

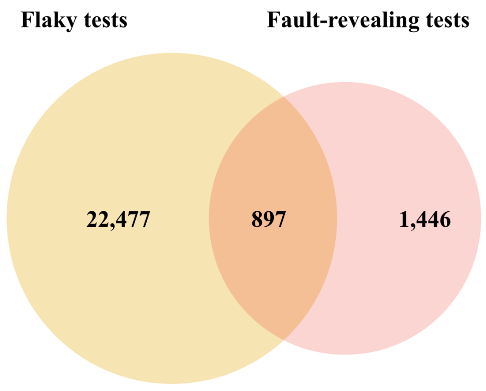


Figure 8.7: Distribution of tests in our dataset. 22,477 tests are exclusively flaky among all builds. 2,343 tests are fault-revealing, among which 1/3 are flaky in other builds.

8.7 Threats to Validity

8.7.1 Internal Validity

The main threat to the internal validity of our study lies in the use of vocabulary-based approaches as predictors of flaky tests and failures. Approaches leveraging other features, *i.e.* dynamic, static, or both, could perform differently. As explained in section 8.4.2, many features are difficult to extract in the case of Chromium (*e.g.* test smells or test dependency graphs) and features relying on code coverage are not considered due to the overheads they introduce and the difficulty of instrumenting the entire codebase. Although this limits our feature set, the same situation appears in many major companies such as Google and Facebook. Nevertheless, our key insight is that many regression faults are discovered by flaky tests, meaning that they would have been missed even by any flaky test detector that correctly considers them as flaky.

As seen in previous sections, most of the research on flakiness prediction focuses on classifying tests. Although in this paper we highlight the need for —and focus on— detecting failures, one may wonder what would be the performance of the studied techniques when aiming at detecting flaky tests (instead of flaky test failures). To this end, we trained a model using our dataset to distinguish flaky from non-flaky tests and found similar results with those reported by the literature, i.e., MCC 0.77 when shuffling data and MCC 0.52 when performing a time-sensitive evaluation. The above result shows that the problem of targeting flaky tests is easier and more predictable. However, as shown by our analysis it is misleading as more than 2/3 of the regression faults are missed by such methods.

8.7.2 External Validity

We show that detecting flaky tests (instead of failures) is harmful as it can miss many regression faults. This is the case for the Chromium project and, while we believe Chromium to be representative of other software systems, we cannot guarantee that findings would generalise to other projects. Similarly, the performance of the different models we report may vary depending on the project. Here, we mainly focus on web/GUI tests and flakiness might have different causes in HTML and Javascript testing compared to other programming languages.

Nevertheless, we believe that flaky tests are useful since developers tend to keep them instead of discarding them. Therefore, flaky test signals should not always be considered as false.

8.7.3 Construct Validity

We assume that all fault-revealing tests in our dataset indeed reveal one or several issues in the code. This is the information reported by the Chromium CI as of today. It is possible that some fault-revealing tests are actually flaky tests as they might not be executed in a sufficient amount of time. However, reruns cannot guarantee that a test is not flaky. As this information is currently used by Chromium's developers and further verification is non-trivial, we rely on it as ground truth for our dataset. Passing tests used as non-flaky tests could also be mislabeled in our dataset. Though, there is a consequent number of passing tests and it is unlikely that many would actually be flaky. Furthermore, to strengthen our confidence in our set of non-flaky tests, we remove from the set of passing tests any tests that were found to be either flaky or fault-revealing in any of the 10,000 builds.

Additionally, it is possible that more than one regression fault is present in the case of several fault-revealing tests that fail in one build. Although this could alter the results we report in RQ1, it would actually strengthen our key message as even more faults could have been missed.

8.8 Conclusion

In this paper, we investigated the utility of existing vocabulary-based flaky test prediction methods in the context of a continuous integration pipeline. To do so, we collected data about 23,374 flaky tests and 2,343 fault-revealing tests composing a dataset of 1.8 million test failures representing the actual development process of more than 10,000 builds corresponding to a period of 9 months. Thus, we empirically evaluated the prediction methods and found similar performance compared to previous studies in terms of precision and recall. Despite the (very) high accuracy to detect flaky test failures, we also found that 76.2% of fault-triggering test failures were misclassified as flaky by the prediction methods, indicating major losses on the fault revelation capabilities of the test suites. Going a step further, we also showed that flaky tests have a strong ability to detect faults, with $1/3$ of all regression faults being revealed by tests that have experienced flaky behaviour at some point in the lifetime of the project under analysis.

These findings motivated the need for failure-focused prediction methods. To this end, we extended our analysis by checking the performance of failure-focused models (trained on failures instead of tests) and found that they result in similar accuracy and fewer false positives. We also found that considering test execution features such as the run duration and the historical flake rate was helpful to increase its ability to discern flaky failures and fault-triggering failures. However, our results still miss a large number of test failures, 42.3%. Therefore, we believe that the current performance is not actionable and that additional research is needed in order to tackle this vastly ignored problem of flaky test failure prediction over flaky tests.

Our future research agenda aims at improving the performance of flaky test failure detection techniques by using additional features and artificial data (augmenting the training data with new positive examples to tackle the class imbalance issues). We also plan to develop failure-cause interpretations for the techniques so that they could be usable by the Chromium developers.

9 Conclusion & Future Work

This chapter presents the overall conclusion of the dissertation and proposes potential research directions.

Contents

9.1 Summary of Contributions

This dissertation aimed at addressing test flakiness, one of the major challenges of modern software testing. It consists of five main scientific contributions: two exploratory studies giving more insights into flakiness in practice, two constructive studies presenting different approaches suggesting new ways of tackling flakiness and a case study evaluating the usefulness of existing prediction techniques in an industrial context. More precisely:

We first performed a grey literature review and interviewed 14 practitioners in order to have a better understanding of the challenges linked with flakiness in the industry. We explore three aspects: the sources of flakiness within the testing ecosystem, the impacts of flakiness and the measures adopted when addressing flakiness. Our analysis showed that, besides the tests and code, flakiness stems from interactions between the system components, the testing infrastructure, and external factors. We also highlighted the impact of flakiness on testing practices and product quality and showed that the adoption of guidelines together with a stable infrastructure are key measures in mitigating the problem. Furthermore, we also identified automation opportunities enabling future research works.

In the second contribution, we performed a replication study of a recently proposed method that predicts flaky tests based on their code vocabulary. We thus replicated the original study on three different dimensions. First, we replicated the approach on the same subjects as in the original study but using a different evaluation methodology, *i.e.* we adopted a time-sensitive selection of training and test sets to better reflect the envisioned use case. Second, we consolidated the findings of the original study by checking the generalisability of the results for a different programming language. Finally, we proposed an extension to the original approach by experimenting with different features extracted from the code under test.

Existing flakiness detection approaches mainly focus on classifying tests as flaky or not and, even when high performances are reported, it remains challenging to understand the cause of flakiness. This part is crucial for researchers and developers that aim to fix it. To help with the comprehension of a given flaky test, the third contribution introduced FlakyCat, the first approach to classify flaky tests based on their root cause category. FlakyCat relies on CodeBERT for code representation and leveraged Siamese networks to train a multi-class classifier. We trained and evaluated FlakyCat on a set of 451 flaky tests collected from open-source Java projects. Our evaluation showed that FlakyCat categorises flaky tests accurately, with an F1 score of 73%. We also investigated the performance of FlakyCat for each category. In addition, to facilitate the comprehension of FlakyCat's prediction, we presented a new technique for CodeBERT-based model interpretability that highlights code statements influencing the categorisation.

To mitigate the effects of flakiness, both researchers and industrial experts proposed strategies and tools to detect and isolate flaky tests. However, flaky tests are rarely fixed as developers struggle to localise and understand their causes. Additionally, developers working with large codebases often need to know the sources of nondeterminism to preserve code quality, *i.e.* avoid introducing technical debt linked with non-deterministic behaviour, and avoid introducing new flaky tests. To aid with these tasks, we proposed with the fourth contribution re-targeting Fault Localisation techniques to the flaky component localisation problem, *i.e.* pinpointing program classes that cause the non-deterministic behaviour of flaky tests. In particular, we employed Spectrum-Based Fault Localisation (SBFL), a coverage-based fault localisation technique commonly adopted for its simplicity and effectiveness. We also utilised other data sources, such as change history and static code metrics, to further improve the localisation. Our results showed that augmenting SBFL with change and code metrics ranks flaky classes in the top-1 and top-5 suggestions, in 26% and 47% of the cases. Overall, we successfully reduced the average number of classes inspected to locate the first flaky class to 19% of the total number of classes covered by flaky tests. Our results also showed that localisation methods are effective in major flakiness categories, such as concurrency and asynchronous waits, indicating their general ability to identify flaky components.

While promising, the actual utility of the methods predicting flaky tests remained unclear since they have not been evaluated within a continuous integration (CI) process. In particular, it remained unclear what is the impact of missed faults, *i.e.* the consideration of fault-triggering test failures as flaky, at different CI cycles. In the last contribution, we applied state-of-the-art flakiness prediction methods at the Chromium CI and checked their performance. Perhaps surprisingly, we find that the application of such methods led to numerous faults missed, which is approximately $^3/_4$ of all regression faults. To explain this result, we analysed the fault-triggering failures and find that flaky tests have a strong fault-revealing capability, *i.e.* they reveal more than $^1/_3$ of all regression faults, indicating inevitable mistakes of methods that focus on identifying flaky tests, instead of flaky test failures. We also found that 56.2% of fault-triggering failures, made by non-flaky tests, are misclassified as flaky. To deal with these issues, we built failure-focused prediction methods and optimized them by considering new features. Interestingly, we found that these methods perform better than the test-focused ones, with an MCC increasing from 0.20 to 0.42. Overall, our findings suggested that future research should focus on predicting flaky test failures instead of flaky tests (to reduce missed faults) and revealed the need for adopting more thorough experimental methodologies when evaluating flakiness prediction methods (to better reflect the actual practice).

9.2 Perspectives

In the following, we discuss potential future research that follows the contributions and ideas presented in this dissertation:

- **Failure Prediction:** As we saw in Chapter 8, one of the real challenges is to detect flaky from relevant test failures, *i.e.* knowing when the developers should investigate issues. While promising, an approach using a limited set of features does not enable efficient performance to reach industrial adoption. Future research should focus on that problem and investigate the use of other features that can be linked with flakiness such as the time of the day tests were executed or the parameters, state and load of the machine running the tests. Many studies in software testing rely on the history of each test to draw conclusions [145], [197]. While we used the flake rate as a feature in our approach, we could think of more elaborative ways to leverage it as done in other studies [103], [205].
- **Machine Learning Interpretability:** Interpretability and explainability of machine learning models are one of the main challenges in AI at the moment. This often restrains the usage of such approaches by practitioners as reliability is often a prerequisite to performance. Nowadays, more and more tools leverage large language models to attempt to solve software engineering problems. However, it would be interesting to investigate, in parallel, techniques to interpret model outcomes and increase the reliability of their usage. Some approaches already exist, like SHAP [160] or Lime [206] but are unsuitable for large language models. As we saw in Chapter 6, this can be useful for developers to know more about the reasons behind the flakiness of a test to debug it and the lack of ground truth for precise root causes opens doors for future research.
- **Practitioners' studies:** In our contributions, we explored different approaches regarding prediction techniques helping with the problem of flakiness. Particularly, Chapters 6 and 7 are constructive studies and aim at assisting developers and researchers in investigating and fixing their flaky tests. Chapter 8 also envisions scenarios where such prediction models would be included in the CI, potentially replacing other techniques like reruns. Another valuable research contribution would be to further conduct surveys and studies directly on developers and practitioners. This is crucial to bridge the gap between academia and the industry and while it often requires lots

of effort to pursue them, it can bring many benefits to better understand concrete needs, limits and to collect new evidence from software development and testing in practice.

Abbreviations

API Application Programming Interface.
AST Abstract Syntax Tree.
AUC Area Under Curve.

CC Cyclomatic Complexity.
CD Continuous Delivery.
CI Continuous Integration.
CPU Central Processing Unit.
CSS Cascading Style Sheet.
CTO Chief Technology Officer.
CUT Code Under Test.

DDU Diversity, Density and Uniqueness.
DOI Depth of Inheritance.
DT Decision Tree.

FL Fault Localisation.
FN False Negative.
FP False Positive.
FPR False Positive Rate.
FSL Few-Shot Learning.
FT Flaky Test.

GLR Grey Literature Review.
GP Genetic Programming.
GUI Graphical User Interface.

HTML Hypertext Markup Language.

IDE Integrated Development Environment.
IRFL Information Retrieval-Based Fault Localisation.
IT Information Technology.

KNN K-Nearest Neighbours.

LOC Lines of Code.

MBFL Mutation-Based Fault Localisation.
MCC Matthew's Correlation Coefficient.
ML Machine Learning.

NFT Non Flaky Test.

OOS Out of Memory.
OS Operating System.
OSS Open Source Software.

QA Quality Assurance.

RAM Random Access Memory.
RF Random Forest.
RTS Regression Test Selection.

SBFL Spectrum-Based Fault Localisation.
SMOTE Synthetic Minority Oversampling Technique.
SQA Software Quality Assurance.
SUT System Under Test.
SVM Support Vector Machine.

TCP Test Case Prioritisation.
TN True Negative.
TP True Positive.

UI User Interface.
URL Uniform Resource Locator.

List of publications and services

Published papers

- S. Habchi, G. Haben, M. Papadakis, *et al.*, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 244–255. DOI: `10.1109/ICST53961.2022.00034`
- G. Haben, S. Habchi, M. Papadakis, *et al.*, "A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests," *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021
- A. Akli, G. Haben, S. Habchi, *et al.*, "Predicting flaky tests categories using few-shot learning," in *Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test*, ser. AST '23, Melbourne, Australia: Association for Computing Machinery, 2023, ISBN: 9781450392860
- S. Habchi, G. Haben, J. Sohn, *et al.*, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 352–363. DOI: `10.1109/ICSME55016.2022.00039`

Papers currently under submission

- G. Haben, S. Habchi, M. Papadakis, *et al.*, *The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci*, 2023. arXiv: `2302.10594 [cs.SE]`

Conferences organisation

BENEVOL 2020 (web chair), ICSME 2021 (web chair)

I acted as a co-reviewer for the following venues

2020: FSE, ISSTA, ICST, ICSME, QRS
2021: ICSE, ICST, MSR, AST, MobileSoft, ICTSS
2022: ICSE, FSE, ISSRE, EMSE (Journal), JSS (Journal)
2023: ICST, AST

I acted as a reviewer for the following venue

2023: STVR (Journal)

List of figures

List of tables

Bibliography

[1] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, “Software development life cycle agile vs traditional approaches,” in *International Conference on Information and Network Technology*, vol. 37, 2012, pp. 162–167 (cit. on p. 2).

[2] R. Jain and U. Suman, “A systematic literature review on global software development life cycle,” *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 1–14, 2015 (cit. on p. 2).

[3] T. Bhuvaneswari and S. Prabaharan, “A survey on software development life cycle models,” *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 5, pp. 262–267, 2013 (cit. on p. 2).

[4] T. Dyba and T. Dingsoyr, “What do we know about agile software development?” *IEEE software*, vol. 26, no. 5, pp. 6–9, 2009 (cit. on p. 3).

[5] A. Koch, *Agile software development.* Artech, 2004 (cit. on p. 3).

[6] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, “Agile software development methods: Review and analysis,” *arXiv preprint arXiv:1709.08439*, 2017 (cit. on p. 3).

[7] K. Beck, M. Beedle, A. Van Bennekum, *et al.*, “Manifesto for agile software development,” 2001 (cit. on p. 3).

[8] *What is ci/cd 101 | all you need to know*, (Accessed on 05/10/2023). [Online]. Available: `https://www.softwaretestingmaterial.com/what-is-ci-cd/` (cit. on p. 3).

[9] M. Fowler and M. Foemmel, *Continuous integration*, 2006 (cit. on p. 4).

[10] M. Shahin, M. A. Babar, and L. Zhu, “Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices,” *IEEE access*, vol. 5, pp. 3909–3943, 2017 (cit. on pp. 4, 74).

[11] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007 (cit. on p. 4).

[12] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010 (cit. on p. 4).

[13] D. Galin, *Software quality assurance: from theory to implementation*. Pearson education, 2004 (cit. on p. 4).

[14] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984 (cit. on p. 4).

[15] P. Tripathy and K. Naik, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011 (cit. on p. 4).

[16] M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, Jan. 2010 (cit. on p. 5).

[17] A. Contan, C. Dehelean, and L. Miclea, "Test automation pyramid from theory to practice," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, IEEE, 2018, pp. 1–5 (cit. on p. 7).

[18] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420–426, 2002 (cit. on p. 7).

[19] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The relationship between test coverage and reliability," in *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, IEEE, 1994, pp. 186–195 (cit. on p. 7).

[20] J. Micco, *The State of Continuous Integration Testing Google*, 2017 (cit. on pp. 8, 10, 54, 74, 98).

[21] M. Harman and P. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, Sep. 2018, pp. 1–23, ISBN: 978-1-5386-8290-6. DOI: `10.1109/SCAM.2018.00009`. [Online]. Available: `https://ieeexplore.ieee.org/document/8530713/` (cit. on pp. 8, 10, 49, 54).

[22] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963 (cit. on p. 8).

[23] M. J. Beheshtian, A. H. Bavand, and P. C. Rigby, "Software batch testing to save build test resources and to reduce feedback time," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2784–2801, 2022. DOI: `10.1109/TSE.2021.3070269` (cit. on p. 8).

[24] A. Najafi, P. C. Rigby, and W. Shang, “Bisecting commits and modeling commit risk during testing,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 279–289 (cit. on p. 8).

[25] G. An and S. Yoo, “Reducing the search space of bug inducing commits using failure coverage,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1459–1462 (cit. on p. 8).

[26] E. Engström, P. Runeson, and M. Skoglund, “A systematic review on regression test selection techniques,” *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010 (cit. on p. 8).

[27] G. Rothermel and M. J. Harrold, “Analyzing regression test selection techniques,” *IEEE Transactions on software engineering*, vol. 22, no. 8, pp. 529–551, 1996 (cit. on p. 8).

[28] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, “An empirical study of regression test selection techniques,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001 (cit. on p. 8).

[29] A. Shi, P. Zhao, and D. Marinov, “Understanding and improving regression test selection in continuous integration,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2019, pp. 228–238 (cit. on p. 8).

[30] A. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in *Proceedings of 1993 15th International Conference on Software Engineering*, 1993, pp. 100–107. DOI: 10.1109/ICSE.1993.346062 (cit. on p. 9).

[31] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 351–360 (cit. on p. 9).

[32] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. Le Traon, “Cerebro: Static subsuming mutant selection,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 24–43, 2022 (cit. on p. 9).

[33] M. Ojdanić, W. Ma, T. Laurent, T. T. Chekam, A. Ventresque, and M. Papadakis, “On the use of commit-relevant mutants,” *Empirical Software Engineering*, vol. 27, no. 5, p. 114, 2022 (cit. on p. 9).

[34] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, pp. 434–487, 2020 (cit. on p. 9).

[35] M. Ojdanic, A. Khanfir, A. Garg, R. Degiovanni, M. Papadakis, and Y. Le Traon, "On comparing mutation testing tools through learning-based mutant selection," in *Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test*, ser. AST '23, Melbourne, Australia: Association for Computing Machinery, 2023 (cit. on p. 9).

[36] W. Ma, T. Laurent, M. Ojdanić, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 394–405 (cit. on p. 9).

[37] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419 (cit. on p. 9).

[38] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012 (cit. on p. 9).

[39] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816 (cit. on p. 9).

[40] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015 (cit. on p. 9).

[41] M. Wen, J. Chen, Y. Tian, *et al.*, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, 2019 (cit. on p. 9).

[42] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016 (cit. on p. 9).

[43] P. S. Leitao-Junior, D. M. Freitas, S. R. Vergilio, C. G. Camilo-Junior, and R. Harrison, "Search-based fault localisation: A systematic mapping study," *Information and Software Technology*, vol. 123, p. 106 295, 2020 (cit. on p. 9).

[44] X. Li, W. Li, Y. Zhang, and L. Zhang, “Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization,” in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180 (cit. on p. 10).

[45] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019 (cit. on p. 10).

[46] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265 (cit. on p. 10).

[47] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, “Do automated program repair techniques repair hard and important bugs?” In *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 25–25 (cit. on p. 10).

[48] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, “Studying the advancement in debugging practice of professional software developers,” *Software Quality Journal*, vol. 25, pp. 83–110, 2017 (cit. on p. 10).

[49] M. Eck, M. Castelluccio, F. Palomba, and A. Bacchelli, “Understanding flaky tests: The developer’s perspective,” *arXiv*, pp. 830–840, 2019 (cit. on pp. 10, 17, 21, 24, 30, 31, 49, 74, 82, 92, 93, 98, 99, 112, 117, 120).

[50] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, “On the impact of flaky tests in automated program repair,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2021, pp. 295–306 (cit. on p. 10).

[51] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman, “Flakime: Laboratory-controlled test flakiness impact assessment,” in *2021 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*, IEEE, 2022 (cit. on pp. 10, 24).

[52] H. Jiang, X. Li, Z. Yang, and J. Xuan, “What causes my test alarm? automatic cause analysis for test alarms in system and integration testing,” in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 712–723, ISBN: 9781538638682. DOI: `10.1109/ICSE.2017.71`. [Online]. Available: `https://doi.org/10.1109/ICSE.2017.71` (cit. on pp. 10, 54).

[53] M. contributors, *Test verification - mozilla | mdn*, `https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification`, (Accessed on 01/12/2021), Mar. 2019 (cit. on pp. 10, 30, 54).

[54] J. Palmer, *Test flakiness – methods for identifying and dealing with flaky tests : Spotify engineering*, `https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/`, (Accessed on 01/12/2021), Nov. 2019 (cit. on pp. 10, 30, 45, 54, 98).

[55] 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`, (Accessed on 01/12/2021), Apr. 2017 (cit. on pp. 10, 30, 54).

[56] J. Micco and A. Memon, *Gtac 2016: How flaky tests in continuous integration - youtube*, `https://www.youtube.com/watch?v=CrzpkF1-VsA`, (Accessed on 01/12/2021), Dec. 2016 (cit. on pp. 10, 11, 54, 98).

[57] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds., IEEE / ACM, 2019, pp. 101–110. DOI: `10.1109/ICSE-SEIP.2019.00019`. [Online]. Available: `https://doi.org/10.1109/ICSE-SEIP.2019.00019` (cit. on pp. 11, 24, 54, 74, 98, 120, 126).

[58] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, Association for Computing Machinery, Nov. 2014, pp. 643–653, ISBN: 9781450330565. DOI: `10.1145/2635868.2635920` (cit. on pp. 16, 17, 21, 24, 30, 31, 54, 55, 58, 62, 67, 76, 81–83, 92, 93, 99, 103, 106, 112, 115).

[59] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "Idflakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322, 2019. DOI: `10.1109/ICST.2019.00038` (cit. on pp. 17, 25, 26, 30, 49, 74, 82, 98, 100, 120).

[60] 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)*, 2022 (cit. on pp. 17, 24).

[61] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An Empirical Study of Flaky Tests in Python," *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021*,

pp. 148–158, 2021. DOI: 10.1109/ICST49551.2021.00026 (cit. on pp. 17, 21, 24, 26, 83, 99, 103, 106).

[62] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," *Proceedings - International Conference on Software Engineering*, pp. 1471–1482, 2020, ISSN: 02705257. DOI: 10.1145/3377811.3381749 (cit. on pp. 17, 21, 76, 103, 106, 112, 116).

[63] T. M. Mitchell *et al.*, *Machine learning*. McGraw-hill New York, 2007, vol. 1 (cit. on p. 17).

[64] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 2021 (cit. on p. 17).

[65] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: Why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017 (cit. on p. 18).

[66] J. A. Hartigan, M. A. Wong, *et al.*, "A k-means clustering algorithm," *Applied statistics*, vol. 28, no. 1, pp. 100–108, 1979 (cit. on p. 18).

[67] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008 (cit. on p. 18).

[68] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010 (cit. on p. 18).

[69] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, pp. 1–13, 2020 (cit. on p. 19).

[70] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, Long Beach, CA, USA: Association for Computing Machinery, 2005, pp. 273–282, ISBN: 1581139934. DOI: 10.1145/1101908.1101949. [Online]. Available: https://doi.org/10.1145/1101908.1101949 (cit. on pp. 20, 26).

[71] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *The proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC 2006, IEEE, 2006, pp. 39–46 (cit. on pp. 20, 103).

[72] W. E. Wong, V. Debroy, R. Gao, and Y. Li, “The dstar method for effective software fault localization,” *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014. DOI: `10.1109/TR.2013.2285319` (cit. on pp. 20, 102, 103).

[73] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “Spectrum-based multiple fault localization,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 2009, pp. 88–99 (cit. on pp. 20, 103).

[74] O. Parry, “A Survey of Flaky Tests,” *ACM transactions on software engineering and methodology*, vol. 31, no. 1, 2021 (cit. on pp. 24, 103, 106, 120, 121).

[75] S. Thorve, C. Sreshtha, and N. Meng, “An empirical study of flaky tests in android apps,” *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 534–538, 2018. DOI: `10.1109/ICSME.2018.00062` (cit. on pp. 24, 30, 31, 55, 106).

[76] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, “An empirical analysis of ui-based flaky tests,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1585–1597 (cit. on pp. 24, 25, 99).

[77] A. M. Memon and M. B. Cohen, “Automated testing of gui applications: Models, tools, and controlling flakiness,” in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 1479–1480 (cit. on p. 24).

[78] N. Hashemi, A. Tahir, and S. Rasheed, “An empirical study of flaky tests in javascript,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2022, pp. 24–34 (cit. on p. 24).

[79] A. Shi, J. Bell, and D. Marinov, “Mitigating the effects of flaky tests on mutation testing,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2019*, New York, New York, USA: ACM Press, 2019, pp. 112–122, ISBN: 9781450362245. DOI: `10.1145/3293882.3330568`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3293882.3330568` (cit. on p. 24).

[80] Y. Qin, S. Wang, K. Liu, and X. Mao, “On the Impact of Flaky Tests in Automated Program Repair,” no. 2, 2021 (cit. on p. 24).

[81] 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 15th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST ’22, 2022 (cit. on pp. 24, 74, 98).

[82] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, “Empirical study of restarted and flaky builds on travis ci,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 254–264 (cit. on p. 24).

[83] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “Deflaker: Automatically detecting flaky tests,” in *Proceedings of the 40th International Conference on Software Engineering - ICSE ’18*, New York, New York, USA: ACM Press, 2018, pp. 433–444, ISBN: 9781450356381. DOI: `10.1145/3180155.3180164`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3180155.3180164` (cit. on pp. 25, 26, 30, 45, 49, 56, 81, 98, 120, 130).

[84] D. Silva, L. Teixeira, and M. D’Amorim, “Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker,” *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 301–311, 2020. DOI: `10.1109/ICSME46990.2020.00037` (cit. on pp. 25, 49, 74, 98).

[85] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root Causing Flaky Tests in a Large-Scale Industrial Setting,” in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’19)*, Beijing, China: ACM Press, 2019, pp. 101–111, ISBN: 9781450362245. DOI: `10.1145/3293882.3330570` (cit. on pp. 25, 30, 31, 47, 49, 54, 58, 74, 116).

[86] C. Ziftci and D. Cavalcanti, “De-flake your tests: Automatically locating root causes of flaky tests in code at google,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Online event: IEEE, 2020, pp. 736–745 (cit. on pp. 25, 74, 116, 120).

[87] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “Ifixflakies : A framework for automatically fixing order-dependent flaky tests,” in *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations ofSoftware Engineering (ESEC/FSE ’19)*, 2019, ISBN: 9781450355728. DOI: `10.1145/3338906.3338925` (cit. on pp. 25, 30, 74, 81, 82, 98).

[88] C. Li, C. Zhu, W. Wang, and A. Shi, “Repairing order-dependent flaky tests via test generation,” in *Proceedings of the 44th International Conference on Software Engineering - ICSE ’22*, ICSE, 2022 (cit. on pp. 25, 74, 82).

[89] J. Morán, C. Augusto, A. Bertolino, C. de la Riva, and J. Tuya, “Flakyloc: Flakiness localization for reliable test suites in web applications,” *J. Web Eng.*, vol. 19, no. 2, pp. 267–296, 2020. DOI: `10.13052/jwe1540-9589.1927`. [Online]. Available: `https://doi.org/10.13052/jwe1540-9589.1927` (cit. on pp. 26, 74).

[90] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, “A practical evaluation of spectrum-based fault localization,” *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009, ISSN: 0164-1212. DOI: `10.1016/j.jss.2009.06.035`. [Online]. Available: `https://doi.org/10.1016/j.jss.2009.06.035` (cit. on p. 26).

[91] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, “Towards a bayesian network model for predicting flaky automated tests,” *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 100–107, 2018. DOI: `10.1109/QRS-C.2018.00031` (cit. on pp. 26, 49, 74, 103, 120).

[92] G. Pinto, B. Miranda, S. Dissanayake, M. D’Amorim, C. Treude, and A. Bertolino, “What is the vocabulary of flaky tests?” *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 492–502, 2020. DOI: `10.1145/3379597.3387482` (cit. on pp. 26, 49, 54–57, 70, 71, 74, 77, 80, 85, 87, 89, 98, 103, 120, 130, 131).

[93] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, “Know Your Neighbor: Fast Static Prediction of Test Flakiness,” *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020. DOI: `10.32079/ISTI-TR-2020/001.Istituto`. [Online]. Available: `https://ieeexplore.ieee.org` (cit. on pp. 26, 49, 71, 120, 131).

[94] 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,” *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021 (cit. on pp. 26, 53, 77, 89, 98, 115, 120, 130, 131, iii).

[95] B. Camara, M. Silva, A. T. Endo, and S. Vergilio, “What is the vocabulary of flaky tests? an extended replication,” in *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 444–454. DOI: `10.1109/ICPC52881.2021.00052`. [Online]. Available: `https://doi.ieeecomputersociety.org/10.1109/ICPC52881.2021.00052` (cit. on pp. 26, 77, 85, 98, 120, 130, 131).

[96] A. Alshammari, C. Morris, M. Hilton, and J. Bell, “Flakeflagger: Predicting flakiness without rerunning tests,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584. DOI: `10.1109/ICSE43902.2021.00140` (cit. on pp. 26, 27, 30, 77, 80, 81, 98, 115, 116, 120, 129, 130).

[97] S. Fatima, T. A. Ghaleb, and L. Briand, “Flakify: A black-box, language model-based predictor for flaky tests,” *IEEE Transactions on Software Engineering*, pp. 1–17, 2022. DOI: `10.1109/TSE.2022.3201209` (cit. on pp. 26, 77, 87, 98, 120).

[98] V. Pontillo, F. Palomba, and F. Ferrucci, “Toward static test flakiness prediction: A feasibility study,” in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, 2021, pp. 19–24 (cit. on p. 26).

[99] 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 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 1381–1392, ISBN: 9781450385626. DOI: `10.1145/3468264.3473931`. [Online]. Available: `https://doi-org.proxy.bnl.lu/10.1145/3468264.3473931` (cit. on pp. 26, 27).

[100] Y. Qin, S. Wang, K. Liu, *et al.*, “Peeler: Learning to effectively predict flakiness without running tests,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Limassol, Cyprus: ICSME, 2022, pp. 257–268. DOI: `10.1109/ICSME55016.2022.00031` (cit. on pp. 26, 27).

[101] D. Olewicki, M. Nayrolles, and B. Adams, “Towards language-independent brown build detection,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2177–2188, ISBN: 9781450392211. DOI: `10.1145/3510003.3510122`. [Online]. Available: `https://doi.org/10.1145/3510003.3510122` (cit. on pp. 26, 120, 131).

[102] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, “Predicting flaky tests categories using few-shot learning,” in *Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test*, ser. AST ’23, Melbourne, Australia: Association for Computing Machinery, 2023, ISBN: 9781450392860 (cit. on pp. 26, 73, iii).

[103] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, “Modeling and ranking flaky tests at apple,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 110–119, ISBN: 9781450371230. DOI: `10.1145/3377813.3381370`. [Online]. Available: `https://doi.org/10.1145/3377813.3381370` (cit. on pp. 26, 54, 115, 120, 126, 129, 144).

[104] K. Herzig and N. Nagappan, “Empirically Detecting False Test Alarms Using Association Rules,” *Proceedings - International Conference on Software Engineering*, vol. 2, pp. 39–48, 2015, ISSN: 02705257. DOI: 10.1109/ICSE.2015.133 (cit. on pp. 27, 115).

[105] 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 *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 244–255. DOI: 10.1109/ICST53961.2022.00034 (cit. on pp. 29, 74, 98, 117, iii).

[106] J. Thomas, *Welcome to the google engineering tools blog! | google engineering tools*, http://google-engtools.blogspot.com/2011/05/welcome-to-google-engineering-tools.html, (Accessed on 02/22/2021), May 2011 (cit. on p. 30).

[107] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, “Detecting flaky tests in probabilistic and machine learning applications,” *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 211–224, 2020. DOI: 10.1145/3395363.3397366 (cit. on pp. 30, 31, 49, 67, 74).

[108] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007. [Online]. Available: http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf (cit. on p. 32).

[109] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” *Information and Software Technology*, vol. 106, pp. 101–121, 2019 (cit. on pp. 32, 50).

[110] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, *Summary of the qualitative results*, https://figshare.com/s/5b252c442fc36e8823cb, (Accessed on 02/24/2021), Feb. 2021 (cit. on pp. 33, 42).

[111] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017 (cit. on p. 33).

[112] S. Adolph, W. Hall, and P. Kruchten, “Using grounded theory to study the experience of software development,” *Empirical Software Engineering*, vol. 16, no. 4, pp. 487–513, 2011 (cit. on p. 33).

[113] S. E. Hove and B. Anda, “Experiences from conducting semi-structured interviews in empirical software engineering research,” in *Software metrics, 2005. 11th ieee international symposium*, IEEE, 2005, 10–pp (cit. on p. 33).

[114] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, “Why and how javascript developers use linters,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 578–589. DOI: `10.1109/ASE.2017.8115668` (cit. on p. 35).

[115] S. Habchi, X. Blanc, and R. Rouvoy, “On adopting linters to deal with performance concerns in android apps,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 6–16. DOI: `10.1145/3238147.3238197` (cit. on p. 35).

[116] B. G. Glaser and J. Holton, “Remodeling grounded theory,” *Historical Social Research/Historische Sozialforschung. Supplement*, pp. 47–68, 2007 (cit. on pp. 35, 49).

[117] C. Schmidt, “The analysis of semi-structured interviews,” *A companion to qualitative research*, pp. 253–258, 2004 (cit. on p. 35).

[118] D. G. Oliver, J. M. Serovich, and T. L. Mason, “Constraints and opportunities with interview transcription: Towards reflection in qualitative research,” *Social forces*, vol. 84, no. 2, pp. 1273–1289, 2005 (cit. on p. 35).

[119] K. Hu, *Test stability - how we make ui tests stable | linkedin engineering*, `https://engineering.linkedin.com/blog/2015/12/test-stability---how-we-make-ui-tests-stable`, (Accessed on 02/24/2021), Dec. 2015 (cit. on pp. 42, 44).

[120] A. Solntsev, *Flaky tests 2 - jvm advent*, `https://www.javaadvent.com/2017/12/flaky-tests-2.html`, (Accessed on 02/24/2021), Dec. 2017 (cit. on pp. 42, 45).

[121] E. Developer, *How to fix flaky tests in your ios/swift codebases - youtube*, `https://www.youtube.com/watch?v=_BOp6WYbq38&ab_channel=EssentialDeveloper`, (Accessed on 02/24/2021), Dec. 2019 (cit. on p. 42).

[122] J. Vimberg, *Effective testing - reducing non-determinism to avoid flaky tests - coding forest*, `https://jivimberg.io/blog/2020/07/27/effective-testing-reducing-non-determinism/`, (Accessed on 02/24/2021), Jul. 2020 (cit. on p. 42).

[123] Testinium, *Flaky tests and how to reduce them - testinium*, `https://testinium.com/blog/flaky-tests-and-how-to-reduce-them/`, (Accessed on 02/24/2021) (cit. on p. 42).

[124] Smartbear, *Managing test flakiness | testcomplete*, `https://smartbear.com/resources/ebooks/managing-ui-test-flakiness/`, (Accessed on 02/24/2021), Jun. 2018 (cit. on pp. 42, 44).

[125] P. Fabio, *Introducing the software testing cupcake (anti-pattern) | thoughtworks*, https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern, (Accessed on 02/24/2021), Jun. 2014 (cit. on p. 42).

[126] S. Pavan, *No more flaky tests on the go team | thoughtworks*, https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team, (Accessed on 02/24/2021), Sep. 2012 (cit. on p. 44).

[127] D. Welter, *Preventing flaky tests from ruining your test suite | gradle enterprise*, https://gradle.com/blog/prevent-flaky-tests/, (Accessed on 02/24/2021) (cit. on p. 44).

[128] T. C. Gang, *Flaky tests - a war that never ends | hacker noon*, https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359, (Accessed on 02/24/2021), Dec. 2017 (cit. on p. 45).

[129] Z. Attas, *Selenium conf 2018 - how to un-flake flaky tests- a new hire's toolkit | confengine - conference platform*, https://confengine.com/conferences/selenium-conf-2018/proposal/6157/how-to-un-flake-flaky-tests-a-new-hires-toolkit, (Accessed on 02/24/2021), Jun. 2018 (cit. on pp. 45, 47).

[130] S. Liviu, *A machine learning solution for detecting and mitigating flaky tests*, en, Library Catalog: medium.com, Oct. 2019. [Online]. Available: https://medium.com/fitbit-tech-blog/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests-c5626ca7e853 (visited on 07/30/2020) (cit. on p. 45).

[131] Fuchsia, *Flaky test policy*, https://fuchsia.dev/fuchsia-src/concepts/testing/test_flake_policy, (Accessed on 02/25/2021), Feb. 2021 (cit. on p. 46).

[132] J. Micco, *Flaky tests at google and how we mitigate them | googblogs.com*, https://www.googblogs.com/flaky-tests-at-google-and-how-we-mitigate-them/, (Accessed on 02/24/2021), May 2016 (cit. on p. 46).

[133] Thethinkingtester, *Think like a tester: Your flaky tests are destroying trust*, http://thethinkingtester.blogspot.com/2019/10/your-flaky-tests-are-destroying-trust.html, (Accessed on 02/24/2021), Oct. 2019 (cit. on p. 46).

[134] A. McPeak, *Flaky tests archives | crossbrowsertesting.com*, https://crossbrowsertesting.com/blog/tag/flaky-tests/, (Accessed on 02/24/2021), Feb. 2018 (cit. on p. 46).

[135] B. Lee, *We have a flaky test problem. flaky tests are insidious. fighting... | by bryan lee | scope | medium*, https://medium.com/scopedev/how-can-we-peacefully-co-exist-with-flaky-tests-3c8f94fba166, (Accessed on 02/24/2021), Jan. 2019 (cit. on p. 47).

[136] C. Wong, J. Meinicke, L. Lazarek, and C. Kästner, “Faster variational execution with transparent bytecode transformation,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 117:1–117:30, 2018. DOI: 10.1145/3276487. [Online]. Available: https://doi.org/10.1145/3276487 (cit. on pp. 49, 50).

[137] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, “Understanding reproducibility and characteristics of flaky tests through test reruns in java projects,” pp. 403–413, 2020, ISSN: 10719458. DOI: 10.1109/issre5003.2020.00045 (cit. on pp. 49, 70, 116).

[138] M. Rehkopf, *What is continuous integration | atlassian*, https://www.atlassian.com/continuous-delivery/continuous-integration, (Accessed on 01/12/2021) (cit. on pp. 54, 74, 120).

[139] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, “The role of replications in Empirical Software Engineering,” *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008, ISSN: 13823256. DOI: 10.1007/s10664-008-9060-1 (cit. on p. 55).

[140] O. S. Gómez, N. Juristo, and S. Vegas, “Understanding replication of experiments in software engineering: A classification,” *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014, ISSN: 09505849. DOI: 10.1016/j.infsof.2014.04.004 (cit. on p. 55).

[141] 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*, vol. 4, no. OOPSLA, pp. 1–29, 2020, ISSN: 2475-1421. DOI: 10.1145/3428270 (cit. on pp. 55, 98).

[142] *Pytest documentation*, https://docs.pytest.org/en/stable/, (Accessed on 03/18/2021) (cit. on p. 60).

[143] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, “REPiR : An Information Retrieval based Approach for Regression Test Prioritization,” *FSE '14, Hongkong*, 2014 (cit. on p. 63).

[144] F. Palomba, A. Zaidman, and A. De Lucia, “Automatic test smell detection using information retrieval techniques,” *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 311–322, 2018. DOI: 10.1109/ICSME.2018.00040 (cit. on p. 63).

[145] M. Azizi and H. Do, "ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2018-October, pp. 144–154, 2018, ISSN: 10719458. DOI: `10.1109/ISSRE.2018.00025` (cit. on pp. 63, 144).

[146] *Permutation importance vs random forest feature importance (mdi) - scikit-learn 0.24.0 documentation*, `https://scikit-learn.org/stable/auto_examples//inspection/plot_permutation_importance.html`, (Accessed on 01/12/2021) (cit. on p. 67).

[147] docs.python.org, *What's new in python 3.1 - python 3.9.1 documentation*, `https://docs.python.org/3/whatsnew/3.1.html#pep-372-ordered-dictionaries`, (Accessed on 01/12/2021), Feb. 2021 (cit. on p. 67).

[148] A. Barnert, *Python - how should i test that dictionaries will always be in the same order? - stack overflow*, `https://stackoverflow.com/questions/50475966/how-should-i-test-that-dictionaries-will-always-be-in-the-same-order/50476093#50476093`, (Accessed on 01/12/2021), May 2018 (cit. on p. 67).

[149] "Gzoltar - automatic testing & debugging using spectrum-based fault localization (sfl)." (Accessed on 01/11/2021). (2020) (cit. on p. 71).

[150] M. d'Amorim. "Damorimrg/msr4flakiness." (Accessed on 01/11/2021). (Apr. 2020) (cit. on p. 71).

[151] A. Memon, Z. Gao, B. Nguyen, *et al.*, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, 2017, pp. 233–242 (cit. on p. 74).

[152] B. Camara, M. Silva, A. Endo, and S. Vergilio, "On the use of test smells for prediction of flaky tests," in *Brazilian Symposium on Systematic and Automated Software Testing*, 2021, pp. 46–54 (cit. on pp. 74, 77, 80, 85, 87, 98, 103, 120, 130).

[153] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 425–436 (cit. on p. 77).

[154] Z. Feng, D. Guo, D. Tang, *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: `10.18653/v1/2020.findings-emnlp.139`. [Online]. Available: `https://aclanthology.org/2020.findings-emnlp.139` (cit. on p. 77).

[155] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?–a structural analysis of pre-trained language models for source code," *arXiv preprint arXiv:2202.06840*, 2022 (cit. on p. 77).

[156] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017 (cit. on pp. 77, 79).

[157] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019 (cit. on p. 80).

[158] *Feature importances with a forest of trees — scikit-learn 1.1.1 documentation*, `https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html`, (Accessed on 06/24/2022) (cit. on p. 80).

[159] 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 *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2022, pp. 93–104 (cit. on pp. 80, 82).

[160] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017 (cit. on pp. 80, 144).

[161] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002 (cit. on p. 80).

[162] K. Costa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, pp. 1–14, 2022. DOI: `10.1109/TSE.2022.3208864` (cit. on pp. 81, 82).

[163] S. Habchi, G. Haben, J. Sohn, *et al.*, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 352–363. DOI: `10.1109/ICSME55016.2022.00039` (cit. on pp. 81, 82, 97, iii).

[164] C. Li and A. Shi, "Evolution-aware detection of order-dependent flaky tests," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 114–125 (cit. on p. 82).

[165] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities," *Educational and psychological measurement*, vol. 30, no. 3, pp. 607–610, 1970 (cit. on p. 83).

[166] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: Synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002 (cit. on pp. 83, 131).

[167] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, “Tsdetect: An open source test smells detection tool,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, 1650
bibrangedash 1654, ISBN: 9781450370431. DOI: `10.1145/3368089.3417921`. [Online]. Available: `https://doi.org/10.1145/3368089.3417921` (cit. on p. 85).

[168] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011 (cit. on p. 85).

[169] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization.,” *Journal of machine learning research*, vol. 13, no. 2, 2012 (cit. on p. 86).

[170] 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,” *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 857–862, 2018. DOI: `10.1145/3236024.3275529` (cit. on p. 98).

[171] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, “Flaky test detection in Android via event order exploration,” in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’21), August 23â•ﬁ28, 2021, Athens, Greece*, vol. 1, Association for Computing Machinery, 2021, pp. 367–378, ISBN: 9781450385626. DOI: `10.1145/3468264.3468584` (cit. on p. 98).

[172] S. Dutta, A. Shi, and S. Misailovic, “Flex: Fixing flaky tests in machine learning projects by updating assertion bounds,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 603–614, ISBN: 9781450385626. [Online]. Available: `https://doi.org/10.1145/3468264.3468615` (cit. on p. 98).

[173] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in github for MSR studies,” in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, IEEE, 2021, pp. 560–564 (cit. on p. 100).

[174] Y. Lou, Q. Zhu, J. Dong, *et al.*, “Boosting coverage-based fault localization via graph-based representation learning,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 664–676, ISBN: 9781450385626. DOI: `10.1145/3468264.3468580`. [Online]. Available: `https://doi.org/10.1145/3468264.3468580` (cit. on pp. 101, 113).

[175] X. Li, W. Li, Y. Zhang, and L. Zhang, “Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 169–180, ISBN: 9781450362245. DOI: `10.1145/3293882.3330574`. [Online]. Available: `https://doi.org/10.1145/3293882.3330574` (cit. on pp. 101, 102, 113).

[176] L. C. Briand, Y. Labiche, and X. Liu, “Using machine learning to support debugging with tarantula,” in *The 18th IEEE International Symposium on Software Reliability (ISSRE’07)*, IEEE, 2007, pp. 137–146 (cit. on p. 101).

[177] M. Papadakis and Y. L. Traon, “Metallaxis-fl: Mutation-based fault localization,” *Journal of Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015 (cit. on p. 101).

[178] S. Hong, B. Lee, T. Kwak, *et al.*, “Mutation-based fault localization for real-world multilingual programs (T),” in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 464–475 (cit. on p. 101).

[179] A. Perez, R. Abreu, and I. HASLab, “Leveraging qualitative reasoning to improve sfl.,” in *IJCAI*, 2018, pp. 1935–1941 (cit. on p. 101).

[180] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016 (cit. on pp. 101, 114).

[181] M. Renieres and S. Reiss, “Fault localization with nearest neighbor queries,” in *Proceedings of the 18th International Conference on Automated Software Engineering*, Oct. 2003, pp. 30–39 (cit. on p. 102).

[182] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, “No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist,” University College London, Tech. Rep. RN/14/14, 2014 (cit. on p. 102).

[183] J. Xuan and M. Monperrus, “Learning to combine multiple ranking metrics for fault localization,” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME 2014, Sep. 2014, pp. 191–200 (cit. on pp. 102, 106).

[184] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, “A learning-to-rank based fault localization approach using likely invariants,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 177–188, ISBN: 978-1-4503-4390-9 (cit. on p. 102).

[185] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, “An empirical study of fault localization families and their combinations,” *IEEE Transactions on Software Engineering*, 2019 (cit. on p. 102).

[186] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, “Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis,” *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, 4:1–4:30, Jul. 2017 (cit. on pp. 102, 114).

[187] J. Sohn and S. Yoo, “Empirical evaluation of fault localisation using code and change metrics,” *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1605–1625, 2021. DOI: `10.1109/TSE.2019.2930977` (cit. on pp. 102, 103, 114).

[188] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, “DEAP: Evolutionary algorithms made easy,” *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012 (cit. on p. 103).

[189] J. A. Jones, M. J. Harrold, and J. T. Stasko, “Visualization for fault localization,” in *Proceedings of ICSE Workshop on Software Visualization*, 2001, pp. 71–75 (cit. on p. 103).

[190] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida: ACM, 2002, pp. 467–477 (cit. on p. 103).

[191] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018. DOI: `10.1109/TSE.2017.2693980` (cit. on p. 103).

[192] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. DOI: 10.1002/spe.2346. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document (cit. on p. 104).

[193] J. Sohn, G. An, J. Hong, D. Hwang, and S. Yoo, "Assisting bug report assignment using automated fault localisation: An industrial case study," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Online event: IEEE, 2021, pp. 284–294 (cit. on pp. 105, 114).

[194] J. Sohn and S. Yoo, "Why train-and-select when you can use them all? Ensemble model for fault localisation," in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO 2019, 2019, pp. 1408–1416 (cit. on p. 105).

[195] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 199–209, ISBN: 9781450305624. DOI: 10.1145/2001420.2001445. [Online]. Available: https://doi.org/10.1145/2001420.2001445 (cit. on pp. 106, 116).

[196] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, 2011 (cit. on p. 107).

[197] M. Wen, J. Chen, Y. Tian, *et al.*, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. DOI: 10.1109/TSE.2019.2948158 (cit. on pp. 113, 144).

[198] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 654–664. DOI: 10.1109/ICSE.2017.66 (cit. on p. 114).

[199] S. Habchi, M. Cordy, M. Papadakis, and Y. L. Traon, "On the use of mutation in injecting test order-dependency," *CoRR*, vol. abs/2104.07441, 2021. arXiv: 2104.07441. [Online]. Available: https://arxiv.org/abs/2104.07441 (cit. on p. 115).

[200] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, *The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci*, 2023. arXiv: 2302.10594 [cs.SE] (cit. on pp. 119, iii).

[201] T. C. D. team, *Chromium/chromium: The official github mirror of the chromium source*, `https://github.com/chromium/chromium`, (Accessed on 07/06/2021), Jul. 2021 (cit. on p. 122).

[202] C. contributors, *The chromium projects*, `https://www.chromium.org/`, (Accessed on 08/17/2021), 2021 (cit. on p. 123).

[203] C. Chen, A. Liaw, L. Breiman, *et al.*, "Using random forest to learn imbalanced data," *University of California, Berkeley*, vol. 110, no. 1-12, p. 24, 2004 (cit. on p. 131).

[204] S.-l. developers, *Sklearn.feature_selection.selectkbest — scikit-learn 1.1.2 documentation*, `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html`, (Accessed on 08/11/2022), Aug. 2022 (cit. on p. 131).

[205] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, "Practical flaky test prediction using common code evolution and test history data," AST '23, (cit. on p. 144).

[206] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144 (cit. on p. 144).