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

Sarra Habchi
University of Luxembourg
sarra.habchi@uni.lu

Guillaume Haben
University of Luxembourg
guillaume.haben@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Maxime Cordy
University of Luxembourg
maxime.cordy@uni.lu

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

***Abstract*—Test flakiness forms a major testing concern. Flaky tests manifest non-deterministic outcomes that cripple continuous integration and lead developers to investigate false alerts. Industrial reports indicate that on a large scale, the accrual of flaky tests breaks the trust in test suites and entails significant computational cost. To alleviate this, practitioners are constrained to identify flaky tests and investigate their impact. To shed light on such mitigation mechanisms, we interview 14 practitioners with the aim to identify (i) the sources of flakiness within the testing ecosystem, (ii) the impacts of flakiness, (iii) the measures adopted by practitioners when addressing flakiness, and (iv) the automation opportunities for these measures. Our analysis shows that, besides the tests and code, flakiness stems from interactions between the system components, the testing infrastructure, and external factors. We also highlight the impact of flakiness on testing practices and product quality and show that the adoption of guidelines together with a stable infrastructure are key measures in mitigating the problem.**

I. 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 [1]. Though, test automation faces major problems with the emergence of flaky tests [2]–[4]. 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 cost. Google reports indicate that 16% of their tests exhibit some flakiness whereas 84% of the transitions from pass to fail involve a flaky test [2]. This entails enormous computational resources since 2-16% of the company's testing budget is dedicated to rerun flaky tests [5].

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 [6]–[9] and industrial systems [10], showing that concurrency and order-dependency are among the main categories of test flakiness. Notably, the study of Eck *et al*. [7] 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 [11], iDFlakies [12], and FlakeFlagger [13] 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 [14] 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 [6]–[10], 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.

II. Related Work

The first study on test flakiness was carried out by Luo *et al*. [6]. 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, many techniques were proposed to detect flaky tests with minimal resources, relying on reruns [12], coverage analysis [11], or static and dynamic test features [13], [15], [16]. Other studies focused on highlighting the effects of flakiness on mutation testing and program repair [17].

Several studies have been conducted to inspect flakiness in industrial contexts. Lam *et al*. conducted two studies [10], [18] about flaky tests at Microsoft. The first study showed that the number of build failures can quickly become significant despite having a low number of flaky tests. Thence, they introduced *RootFinder*, a tool that identifies the root causes of flaky tests by analysing differences in test logs and spotting suspect method calls. In their 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. Leong *et al*. [19] studied flaky tests at Google and found that more than 80% of test output transitions are caused by flakiness. At Apple, Kowalczyk *et al*. [20] introduced a flakiness scoring system and showed its ability to reduce flakiness by 44%.

Eck *et al*. [7] 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. In this study, we leverage a different 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.

III. Preliminary analysis: Grey Literature Review

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.

a) Search: We followed the recommendations of Kitchenham and Charters [21], for the reviewing process in general, and the guidelines of Garousi *et al*. [22] 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.

b) 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*. [22], 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 [23]. 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 IV). The results of this analysis are presented in Table II and will be discussed in Section V. 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.

IV. Interviews & 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 [24] based on classic Grounded Theory concepts [25]. In this section, we explain our implementation of this approach from the interview design to the analysis of the results.

A. 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*. [26]. 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.

*a) **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?

*b) **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?

*c) **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.

1) 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 [27], [28]. 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 [29]. Indeed, the three last interviews did not lead to any changes in our analytical template and only provided new formulations for existing categories.

Table I 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.

TABLE I
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

B. Analysis

As our study builds on semi-structured interviews, we relied on the strategy proposed by Schmidt *et al.* [30]. 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.

a) 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 [31]. 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.

b) 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.

c) 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.

d) 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.

V. Study Results

A. RQ1: *Where can we locate flakiness?*

1) ***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.

2) ***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).

3) ***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.

4) ***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).

5) ***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.

6) ***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"*.

7) ***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).

a) ***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.

B. RQ2*: How do practitioners perceive the impact of flakiness?*

1) ***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).

2) ***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).

3) ***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).

4) ***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"*.

5) ***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).

a) ***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.

C. RQ3*: How do practitioners address flaky tests?*

Table II 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 [23].

1) Prevention measures: This represents all proactive practices that aim to prevent the introduction of test flakiness.

a) ***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 [32]–[34]. Some articles also stressed the importance of using containers to ensure that the testing environment is clean when the tests are run [35]. 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).

b) ***Define testing guidelines****:* One guideline that was recurrently mentioned is following the testing pyramid principles [35]–[37]. 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 [38], where the number of GUI tests, which are a main source of flakiness, is exaggerated. Interestingly, only two of our interviewees (P11 & P14) 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.

c) ***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 [32], [37]. 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).

d) ***Customise the testing framework****:* Sudarshan *et al*. [39] 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 [32].

2) Detection measures: This category groups all actions taken by developers to identify flaky tests.

a) ***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"*.

b) ***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 [40]. In the case of GUI tests, practitioners rely particularly on the screenshots recorded during the test run [33], [41], [42]. 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

TABLE II
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 the testing activity.	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 connections, services, and 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 failure or a flaky one.	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 failures 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 and 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 from the suite.	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%

make rerunning the same test in the exact same conditions impossible.

*c) **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*. [43] 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"*.

*d) **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 [43], [44]. 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).

*e) **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 [11]. However, P14 explains that, due to hidden dependencies between projects, this technique is not always effective.

3) Treatment measures: This presents actions taken by practitioners to deal with flay tests that manifested.

*a) **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.

*b) **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).

*c) **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 [45]. 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).

d) ***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 [45]–[47]. 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).

e) ***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 [48]. 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.

4) Support measures: This includes actions that are likely unrelated to test flakiness but are critical for addressing flaky tests.

a) ***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.

b) ***Establish testing workflows****:* For complex software systems, practitioners can design advanced testing paths that organise tests based on their criticality for the integration [42], [49]. 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).

c) ***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*. [10] 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.

D. RQ4: *How could mitigation measures be improved with automation tools?*

1) ***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.

2) ***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.

3) ***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.*, [50]). 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.

*a) **Discussion:**:* Our results confirm previous observations [7], [10], [51] 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 [11], [12], [15], [52], [53]. 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 [54], 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 [55] and FLASH [9], seem very promising if we consider the role of external factors on flakiness.

VI. Threats to Validity

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 [29].

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*. [22], 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.

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.

VII. Implications

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.

• The establishment of simple 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 [50] 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 indicator 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.

Acknowledgment

This work is supported by the Luxembourg National Research Funds (FNR) through the TestFast Project, *ref. 12630949*. Sarra Habchi and Guilliame Haben are supported by PayPal.

REFERENCES

[1] 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, May 2011, (Accessed on 02/22/2021).

[2] 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, April 2017, (Accessed on 01/12/2021).

[3] M. contributors, "Test verification - mozilla — mdn," https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification, March 2019, (Accessed on 01/12/2021).

[4] 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/, November 2019, (Accessed on 01/12/2021).

[5] A. Micco, John & Memon, "Gtac 2016: How flaky tests in continuous integration - youtube," https://www.youtube.com/watch?v=CrzpkF1-VsA, December 2016, (Accessed on 01/12/2021).

[6] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. Hong Kong, China: Association for Computing Machinery, Nov. 2014, pp. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[7] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, Aug. 2019, pp. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[8] 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.

[9] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 211–224. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397366

[10] 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.

[11] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 433–444, iSSN: 1558-1225.

[12] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Apr. 2019, pp. 312–322, iSSN: 2159-4848.

[13] 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)*. IEEE, 2021, pp. 1572–1584.

[14] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: a framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, Aug. 2019, pp. 545–555. [Online]. Available: https://doi.org/10.1145/3338906.3338925

[15] 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.

[16] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 219–229.

[17] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair," *CoRR*, vol. abs/1912.03197, 2019. [Online]. Available: http://arxiv.org/abs/1912.03197

[18] W. Lam and K. Muşlu, "A study on the lifecycle of flaky tests," p. 12, 2020.

[19] 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. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00019

[20] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at apple," *Proceedings - International Conference on Software Engineering*, pp. 110–119, 2020.

[21] 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

[22] 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.

[23] Authors, "Summary of the qualitative results," https://figshare.com/s/5b252c442fc36e8823cb, February 2021, (Accessed on 02/24/2021).

[24] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[25] 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.

[26] 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, pp. 10–pp.

[27] 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.

[28] 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.

[29] B. G. Glaser and J. Holton, "Remodeling grounded theory," *Historical Social Research/Historische Sozialforschung. Supplement*, pp. 47–68, 2007.

[30] C. Schmidt, "The analysis of semi-structured interviews," *A companion to qualitative research*, pp. 253–258, 2004.

[31] 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.

[32] 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, December 2015, (Accessed on 02/24/2021).

[33] A. Solntsev, "Flaky tests 2 - jvm advent," https://www.javaadvent.com/2017/12/flaky-tests-2.html, December 2017, (Accessed on 02/24/2021).

[34] E. Developer, "How to fix flaky tests in your ios/swift codebases - youtube," https://www.youtube.com/watch?v=_BOp6WYbq38&ab_channel=EssentialDeveloper, December 2019, (Accessed on 02/24/2021).

[35] 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/, July 2020, (Accessed on 02/24/2021).

[36] 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).

[37] Smartbear, "Managing test flakiness — testcomplete," https://smartbear.com/resources/ebooks/managing-ui-test-flakiness/, June 2018, (Accessed on 02/24/2021).

[38] P. Fabio, "Introducing the software testing cupcake (anti-pattern) — thoughtworks," https://www.thoughtworks.com/insights/blog/

introducing-software-testing-cupcake-anti-pattern, June 2014, (Accessed on 02/24/2021).
[39] S. Pavan, "No more flaky tests on the go team — thoughtworks," https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team, September 2012, (Accessed on 02/24/2021).
[40] D. Welter, "Preventing flaky tests from ruining your test suite — gradle enterprise," https://gradle.com/blog/prevent-flaky-tests/, (Accessed on 02/24/2021).
[41] T. C. Gang, "Flaky tests - a war that never ends — hacker noon," https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359, December 2017, (Accessed on 02/24/2021).
[42] 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, June 2018, (Accessed on 02/24/2021).
[43] 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 02/25/2021).
[44] S. Liviu, "A machine learning solution for detecting and mitigating flaky tests - engineering fitness," https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests/, (Accessed on 02/25/2021).
[45] Fuchsia, "Flaky test policy," https://fuchsia.dev/fuchsia-src/concepts/testing/test_flake_policy, February 2021, (Accessed on 02/25/2021).
[46] 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/, May 2016, (Accessed on 02/24/2021).
[47] Thethinkingtester, "Think like a tester: Your flaky tests are destroying trust," http://thethinkingtester.blogspot.com/2019/10/your-flaky-tests-are-destroying-trust.html, October 2019, (Accessed on 02/24/2021).
[48] A. McPeak, "flaky tests archives — crossbrowsertesting.com," https://crossbrowsertesting.com/blog/tag/flaky-tests/, February 2018, (Accessed on 02/24/2021).
[49] 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, November 2019, (Accessed on 02/24/2021).
[50] C. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 117:1–117:30, 2018. [Online]. Available: https://doi.org/10.1145/3276487
[51] 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.
[52] 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.
[53] 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. [Online]. Available: https://ieeexplore.ieee.org
[54] 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, 2018, pp. 1–23.
[55] 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.