Surveying the Developer Experience of Flaky Tests

Owain Parry
University of Sheffield
UK

Gregory M. Kapfhammer
Allegheny College
USA

Michael Hilton
Carnegie Mellon University
USA

Phil McMinn
University of Sheffield
UK

ABSTRACT

Test cases that pass and fail without changes to the code under test are known as *flaky*. The past decade has seen increasing research interest in flaky tests, though little attention has been afforded to the views and experiences of software developers. In this study, we utilized a multi-source approach to obtain insights into how developers define flaky tests, their experiences of the impacts and causes of flaky tests, and the actions they take in response to them. To that end, we designed a literature-guided developer survey that we deployed on social media, receiving 170 total responses. We also searched on StackOverflow and analyzed 38 threads relevant to flaky tests, offering a distinct perspective free of any self-reporting bias. Using a mixture of numerical and thematic analyses, this study revealed a number of findings, including (1) developers strongly agree that flaky tests hinder continuous integration; (2) developers who experience flaky tests more often may be more likely to ignore potentially genuine test failures; and (3) developers rate issues in setup and teardown to be the most common causes of flaky tests.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Software Testing; Flaky Tests; Qualitative Research.

ACM Reference Format:
Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513037

1 INTRODUCTION

Test cases that pass and fail without changes to the code under test are called *flaky* and are prevalent in industry [24, 32]. Flaky tests are a major snag for software developers because they disrupt continuous integration [14], harm productivity [29, 33], and lead to a loss of confidence in testing [45]. They are also a persistent problem in research, limiting the applicability of state-of-the-art techniques for test selection and prioritization [31, 36, 48].

Despite an increasing volume of studies on test flakiness [21, 30, 40, 46], there is as yet little focus on the views and experiences of developers. Since flaky tests are primarily a developer problem, there is an underutilized opportunity to acquire valuable insights from those who experience them first-hand. Where previous studies do exist, they focus on specific organizations and developers' self-reported experiences [17, 23], two potential sources of bias [13].

In this paper, we examine multiple sources to understand how developers define and react to flaky tests and their experiences of the impacts and causes. We consulted both published and gray literature to inform the design of a developer survey that we deployed on social media as broadly as possible, receiving 170 responses.

In addition to the survey, we searched StackOverflow and filtered the results to produce a dataset of 38 threads. We performed thematic analysis [11] on the questions and accepted answers in these threads to gain insights into the flaky tests for which developers require assistance to diagnose and repair. Through this unique perspective, we were able to identify themes regarding additional causes and actions that were not revealed by the developer survey.

Some of our findings support previous literature while several others were unanticipated. For example, we found that participants who experience flaky tests more often may be more likely to ignore potentially genuine test failures, supporting the position of experts such as Martin Fowler [19]. We also found that participants rated asynchronicity and concurrency as only the fourth and fifth most common causes of flaky tests respectively, despite studies reporting these to be the most common [17, 30, 40]. Finally, we make all our data and artifacts publicly available in our replication package [6].

In summary, the main contributions of this study are as follows:
Contribution 1: Developer survey: We designed a developer survey based on previous literature and received 170 responses. Through numerical and thematic analysis, we identify alternative definitions of flaky tests, the most significant impacts of flaky tests, the most frequent causes of flaky tests, and the most common actions developers perform in response to flaky tests (See §2.1).
Contribution 2: StackOverflow threads: Through thematic analysis of our dataset of 38 StackOverflow threads, we offer a unique insight into the causes of flaky tests experienced by developers and the strategies that they suggest to repair them, independent of what they self-reported in our developer survey (See §2.2).
Contribution 3: Findings and recommendations: We surface a range of findings that support previous literature and some that were more unforeseen. From these, we offer actionable recommendations for both software developers and researchers (See §4).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513037

2 METHODOLOGY

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

RQ1: Definition: *How do developers define flaky tests?*
RQ2: Impacts: *What impacts do flaky tests have on developers?*
RQ3: Causes: *What causes the flaky tests experienced by developers?*
RQ4: Actions: *What actions do developers take against flaky tests?*

To answer these RQs, we used a multi-source study consisting of a developer survey, with both closed- and open-ended questions, and an analysis of StackOverflow threads. We performed numerical analysis on the closed-ended survey questions and thematic analysis [11] on the open-ended questions and the StackOverflow threads.

Before conducting our study, we received ethical approval for the developer survey from the University of Sheffield. We include our participant information sheet in our replication package [6].

2.1 Developer Survey

We designed a survey of 11 questions for software developers. Some presented a list of prepared statements and asked participants to respond to them in a closed-ended fashion. To ensure their relevance, we reviewed published research and other gray literature to determine these statements. We disseminated the survey on Twitter and LinkedIn, specifically asking for developer participants. We also circulated the survey among Sheffield Digital, a regional technology business forum [7]. Our 11 survey questions are as follows:

SQ1: *A flaky test is a test case that can both pass and fail without any changes to the code under test. Do you agree with this definition?* Participants could indicate that they agreed or disagreed. This is a common definition, though it is not universal. Vahabzadeh et al. [47] only considered test cases whose non-determinism was caused by timing or concurrency to be flaky tests. On the other hand, Shi et al. [41] included test cases with inconsistent coverage in their study on mitigating the effects of flaky tests on mutation testing.

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

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

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

SQ4.1: Reliability: *Flaky tests reduce the reliability of testing.* Vahabzadeh et al. [47] categorized 443 bug reports regarding test cases. They found that 97% caused the test to fail without indicating a bug, i.e., they were *false alarms*. They categorized 53% of these as either flaky tests, resource mishandling, or caused by factors in the execution environment. According to our proposed definition in **SQ1**, we also consider the latter two categories as flaky tests.

SQ4.2: Efficiency: *Flaky tests reduce the efficiency of testing.* A specific category of flaky tests, known as *order-dependent (OD) tests* [26, 43, 50], are influenced by previously executed test cases. Techniques to improve the efficiency of testing by reordering, reducing or splitting-up the test suite are unsound when OD tests are present. For instance, in the test suites of 11 Java modules, Lam et al. [27] found that 23% of OD tests failed after they applied test case prioritization, 24% after test case selection, and 5% after parallelization.

SQ4.3: Productivity: *Flaky tests lead to a loss of productivity.* John Micco [33] explained that developer productivity relies on the capability of test cases to identify genuine issues in a timely and reliable manner. Flaky tests are unreliable and therefore could harm the productivity of the developers who experience them as well as those who rely on the productivity of those developers [29].

SQ4.4: Confidence: *Flaky tests lead to a loss of confidence in testing.* Given how flaky tests may manifest as a false alarm [47], there is a danger that developers will lose confidence in testing if they continuously experience flaky tests. This could lead to a culture of ignoring tests, which may cause genuine bugs to go unnoticed [45].

SQ4.5: CI: *Flaky tests hinder continuous integration (CI).* Durieux et al. [14] analyzed over 75 million build logs on Travis CI. They found that 47% of previously failing builds that were manually restarted by a developer subsequently passed. Since no changes were involved, these builds may have failed due to flaky tests.

SQ4.6: Ignore: *Flaky tests make it more likely for you to ignore (potentially genuine) test failures.* Martin Fowler [19] explained how developers may be tempted to ignore flaky test failures. He explained that if a test suite contains too many flaky tests, developers could lose the discipline to ignore just the flaky failures. Rahman et al. [39] found that ignoring test failures, flaky or not, was associated with a higher volume of crashes due to missed bugs.

SQ4.7: Reproduce: *It is difficult to reproduce a flaky test failure.* Lam et al. [28] described how, upon encountering a failing test, a developer might rerun it in isolation from the rest of the test suite in order to reproduce the failure and debug the code under test. They reran in isolation, for 4,000 times each, the 107 flaky tests from 26 Java modules, only reproducing the failures of 57 and concluding that this may be ineffective at reproducing flaky test failures.

SQ4.8: Differentiate: *It is difficult to differentiate between a test failure due to a genuine bug and a test failure due to flakiness.* Lamyaa Eloussi [18] remarked how flaky tests are a source of wasted time, particularly during regression testing, where flaky test failures may appear linked with a commit but can actually be unrelated.

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

SQ5.1: Waiting: *Not correctly waiting for the results of asynchronous calls to become available.* This cause has been presented in numerous studies and is widely agreed upon by researchers to be one of the leading causes of flaky tests [17, 25, 30, 40]. For example, a flaky test that spawns a new process to perform an operation but does not wait for the process to finish falls under this category.

SQ5.2: Concurrency: *Synchronization issues between multiple threads interacting in an unsafe or unanticipated manner (e.g., data races, atomicity violations, and deadlocks).* Like **SQ5.1**, studies point to this category as being very common. Eck et al. [17] explained that flaky tests caused by *local* thread synchronization issues belong in this category, while synchronization issues with *remote* resources, such as web servers or external processes, would be **SQ5.1**.

SQ5.3: Setup/teardown: *Tests not properly cleaning up after themselves or failing to set up their necessary preconditions.* Many studies identified OD tests to be a very prevalent category of flaky tests [17, 21, 26, 30]. Bell et al. [9] suggested that one cause may be the burden of writing correct setup and teardown methods, executed by a test runner before and after the main body of a test case. Shi et al. [43] differentiated between *victims*, an OD test that fails if executed after a *polluter* test case, and *brittles*, an OD test that only passes if executed before a *state-setter*. In the former, the victim does not perform proper setup and/or the polluter does not perform proper teardown. The latter instance is similar but reversed.
SQ5.4: Resources: *Improper management of resources (e.g., not closing a file or not deallocating memory).* The specific case of failing to release acquired resources (i.e., a resource leak) has been identified at a generally lower prevalence than the preceding categories in previous research [17, 21, 25, 30]. Bearing some similarities to **SQ5.3**, the test that improperly manages the resource may not be the test case that is flaky, but rather a subsequently executed test.
SQ5.5: Network: *Dependency on a network connection.* Any test case that requires a network connection will inevitably be flaky since infrastructure issues or periods of high traffic may cause the test case to fail. Several empirical studies have described this particular cause, with varying degrees of prevalence [17, 25, 30, 46].
SQ5.6: Random: *Not accounting for all the possible outcomes of random data generators or code that uses them.* Test cases that use random data, or cover code that utilizes randomization, can become flaky for a variety of reasons. One reason is that it may be difficult for developers to approximate test oracles, such as the appropriate range of output values in assertion statements [17, 35]. This is a particular problem for machine learning applications [15, 16].
SQ5.7: Time/date: *Reliance on the local system time/date.* Test cases that depend on time and date are fraught with difficulty, such as inconsistencies in representation and precision across systems as well as timezone conversion issues [17, 21, 25, 30].
SQ5.8: Floating point: *Inaccuracies when performing floating point operations.* Given their limited precision and other idiosyncrasies, floating point comparisons can sometimes produce unexpected results. In the context of flaky tests, previous work generally considered this specific cause to be quite rare [17, 25, 30].
SQ5.9: Unordered: *Assuming a particular iteration order for an unordered collection-type object (e.g., sets).* This is a special case of a general cause pertaining to assumptions regarding the implementations of non-deterministic program specifications [22, 42, 49].
SQ5.10: Unknown: *Reasons that cannot be precisely determined.* Finally, developers could indicate how often they had encountered flaky tests whose cause they could not precisely identify.

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

SQ7: *After identifying a flaky test, how often do you...* This question offered a list of actions and participants could rate how often they perform them on a four-point scale. They were as follows:
SQ7.1: No action: *Take no action.* Quite simply, a developer may choose to take no action when encountering a flaky test.
SQ7.2: Re-run: *Re-run the build.* Perhaps the most straight-forward action, a developer may just restart the failing build and hope that the flaky test passes this time. In their study of Travis CI build logs, Durieux et al. [14] found this to be a common practice.
SQ7.3: Document: *Document and defer (e.g., submit an issue/bug report).* A developer may not have the time to immediately repair a flaky test and may choose to document it for attention later. For example, they could raise an issue in a GitHub repository.
SQ7.4: Delete: *Delete the test.* Another straight-forward action is to permanently remove the flaky test from the test suite. Recounting on his experiences at Facebook, Kent Beck remarked how it was routine to delete non-deterministic test cases [12].
SQ7.5: Quarantine: *Quarantine the test.* Martin Fowler [19] advised that flaky tests should be *quarantined* from the main test suite into a dedicated test suite that is understood by the development team to be unreliable. He advised that developers should keep the quarantined test suite small by promptly fixing flaky tests. Otherwise, there is a danger of flaky tests being forgotten about and the whole process becoming equivalent to just deleting test cases.
SQ7.6: Mark skip: *Mark the test to be skipped or as an expected failure (e.g., xfail).* Many testing frameworks allow test cases to be marked as *skipped*, meaning they are not deleted from the test suite but are not executed either. Alternatively, some frameworks, such as `pytest`, allow test cases to be marked as *expected failures* or *xfails*. This signals to the testing framework that they are expected to fail, in which case they should not fail the entire test suite.
SQ7.7: Mark re-run: *Mark the test to be automatically repeated (e.g., by using the flaky plugin for pytest).* Often via the support of plugins, some testing frameworks allow test cases to be marked such that they are repeated some number of times upon failure. One example of such a plugin is `flaky` for `pytest` [10].
SQ7.8: Repair: *Attempt to repair the flakiness.* Finally, a developer may attempt to repair the underlying cause of the flaky test rather than just mitigating it with one of the previous actions.

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

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

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

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

2.2 StackOverflow Threads

We analyzed StackOverflow threads where a developer asked for help addressing one or more flaky tests. We selected StackOverflow specifically due to its widespread popularity and its use in previous software engineering research [34]. This analysis adds further depth to our answer for **RQ3** by considering the causes of flaky tests that developers ask for help with, as opposed to the causes they report as the most common. It is also free of any self-reporting bias that may

result from the survey [13]. For **RQ4**, this analysis offers insights into *how* developers repair flaky test cases, since our survey only asks participants *how often* they attempt to do so.

A thread on StackOverflow consists of a single question followed by answers. A user can indicate that an answer has addressed their question by *accepting* it. To find relevant threads, we used the website's search feature. The query we used was "`flaky test hasaccepted:yes`". The latter part of the query is a search operator that only matches threads where the user who asked the question has accepted an answer. This is to increase the probability that we can identify a recommended course of action for **RQ4**.

From the search results, we created a dataset of relevant threads. We only included threads where the question was specifically asking about the cause of one or more flaky tests and/or how to repair them. We excluded threads where the question was more tangential, such as asking how to handle flaky tests in a specific testing framework [3]. This is because such threads do not present causes or possible repairs and are therefore of no use for addressing **RQ3** or **RQ4**.

2.3 Analysis

We designed **SQ4**, **SQ5**, and **SQ7** to be answered using a four-point Likert scale, quantifying agreement in the case of **SQ4** and frequency for **SQ5** and **SQ7**. Each of these questions asked participants to respond to a list of prepared statements. For each statement, we assigned a score between 0 and 3 to each point on the Likert scale. As an example, for each statement of **SQ4**, participants could select *Strongly disagree*, *Agree*, *Disagree*, or *Strongly disagree*, corresponding to a score of 0, 1, 2, or 3 for that statement, respectively. For each question, we calculated the mean score of each statement across all participants and four specific populations. The first two populations were participants who said they experienced flaky tests on at least a monthly basis and those who experienced them less frequently (see **SQ3**). We chose to split the participants by this criterion since those who frequently experience flaky tests may have different views to those who experience them rarely [19].

The final two populations were participants who said they had at least 13 years of software development experience and those who had fewer (see **SQ10**). We made this split because, when compared to participants with less experience, those with more may be better at accurately diagnosing the causes of flaky tests and may be more likely to take certain actions to address them. We excluded any respondent from the analysis of a particular question if they did not respond to all of that question's statements. For each question, we calculated the ranks of every statement, based on mean score, to quantify the most significant impacts in the case of **SQ4** and the most common causes and actions for **SQ5** and **SQ7**, respectively.

We performed *inductive thematic analysis* [11] on the responses to the open-ended survey questions (**SQ2**, **SQ6**, **SQ8**, and **SQ11**) and the StackOverflow threads. For each survey question, all four authors met and discussed each response. We split responses containing logical connectives such as "and" and "or" into their atomic components. We then assigned one or more labels or *codes* to each response, representing its key concepts. By collaboratively performing the coding, we minimized the impact of any individual biases and ensured our coding was as consistent as possible. From these codes we derived a set of themes, representing the definition of flaky tests, their causes, and developers' actions against them for **SQ2**, **SQ6** and **SQ8**, respectively. For **SQ11**, the themes represent more general insights. We performed a very similar procedure for the StackOverflow threads. Next, we assigned themes regarding causes and actions to each thread in two separate sessions. Finally, we encountered several accepted answers prescribing multiple actions, in which case we assigned them to multiple action themes.

2.4 Threats to Validity

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

Replicability: *Can others replicate our results?* Our numerical analysis is straightforward to replicate. We make the response data from our developer survey and our Python script for performing our numerical analysis available as part of our replication package [6]. In general, thematic analysis is more challenging to replicate. Nonetheless, we include in our replication package the spreadsheets we used to facilitate our collaborative thematic analysis.

Construct: *Are we asking the right questions?* The construction of our study can bias our results, as in any study. To attain the highest quality of results possible, we designed a multi-source approach. Our numerical analysis of the closed-ended survey questions provides broad, high-level insights. Our thematic analysis of the open-ended questions offers a more specific, but much more detailed, understanding of developers' experiences. Finally, our analysis of the StackOverflow threads provides an alternative perspective, free of any potential self-reporting bias [13]. We selected these three components to collect inherently different but complementary data, thereby giving a more complete understanding of flaky tests.

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

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

3 RESULTS

We received 170 responses to our survey. Figure 1 shows the results of **SQ3** and **SQ9**. For **SQ3**, just over half of the participants reported that they observe flaky tests on at least a monthly basis. This shows that flaky tests are a frequent phenomenon, especially given that 23 reported experiencing them daily. For **SQ10**, just under half said

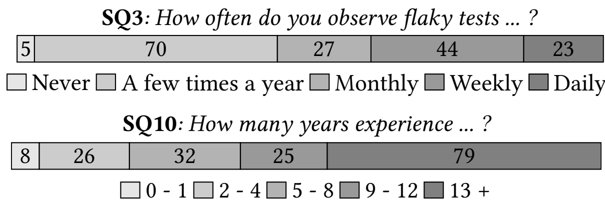

Figure 1: Results for SQ3 and SQ10.

they had 13 years or more of software development experience. For **SQ9**, the top three programming languages were JavaScript, Python, and Java. The distribution roughly corresponds to the most popular languages according to the 2021 StackOverflow developer survey [8]. This reassures us that our participant population is generally representative of the wider community of developers.

After performing our search on StackOverflow, we found 169 threads. We carefully examined each one and narrowed them down to the 38 that are relevant according to the criteria in Section 2.2.

We now answer each of our research questions using the results of our analysis of the responses to our survey and the StackOverflow threads. See Table 1 for the results of our thematic analysis for **SQ11**.

RQ1: Definition. Of the 169 respondents who answered **SQ1**, 6.5% disagreed with our proposed definition. In order of prevalence, the themes following our thematic analysis for **SQ2** are:

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

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

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

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

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

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

RQ2: Impacts. The top third of Table 2 shows the mean scores and ranks of each impact statement. For all participants, **SQ4.5** scored the highest. This indicates that developers strongly agree with the notion that flaky tests hinder CI. Second and third were **SQ4.3** and **SQ4.2**, regarding losses to productivity and the efficiency of testing, respectively. The lowest scoring impact was **SQ4.8**, which, as illustrated by the distribution bar, was the most evenly split between agreement and disagreement. This suggests that differentiating between a true test failure and a spurious failure due to flakiness is relatively straightforward for some developers.

The most significant difference in mean score between participants who experienced flaky tests on at least a monthly basis and those who did not was for **SQ4.6**. This indicates that developers who experience flaky tests more often could be more likely to ignore potentially genuine test failures. Beyond that, the scores are similar, with both scoring **SQ4.5** the highest and **SQ4.8** the lowest. The scores between the participants with at least 13 years of development experience and those with fewer are also similar.

In **SQ11t2**, participants expressed anger or frustration at flaky tests. P96 said they "they're very annoying". Along with **SQ4.4** and **SQ4.6**, this further evidences the psychological cost of flaky tests.

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

RQ3: Causes. The middle third of Table 2 shows the mean scores of each cause we proposed for **SQ5**. The cause with the highest score across all participants was **SQ5.3**. This suggests that improper setup and teardown is the most frequent cause of flaky tests. Flakiness caused by network issues (**SQ5.5**) and unknown reasons (**SQ5.10**) had the second and third highest scores, respectively. This indicates that the causes of many flaky tests go undiagnosed by developers. The lowest scoring was **SQ5.8** concerning floating point issues.

Comparing the participants who experienced flaky tests at least monthly to those who did not, there is agreement that **SQ5.3**, **SQ5.5**, and **SQ5.8** were the first, second, and least most common causes,

Table 1: Themes of the responses for SQ11 regarding miscellaneous insights about flaky tests, in order of prevalence.

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

Table 2: Results of the numerical analysis of the responses for SQ4, SQ5, and SQ7. The five "Mean Score (Rank)" columns are the mean scores of each impact, cause, or action for the four specific populations and all participants (All). In each case, the ranks in descending order of mean score are in parentheses. The final column visualizes the distribution of responses.

	Mean Score (Rank)					
	≥ Monthly (SQ3)		≥ 13 Years (SQ10)			
Question	**Yes**	**No**	**Yes**	**No**	**All**	**Distribution (All)**
RQ2: Impacts						Strongly disagree / Disagree / Agree / Strongly agree
SQ4.1: Reliability	2.43 (4)	2.47 (2)	2.49 (4)	2.41 (4)	2.45 (4)	
SQ4.2: Efficiency	2.53 (3)	2.38 (4)	2.52 (2)	2.42 (3)	2.47 (3)	
SQ4.3: Productivity	2.58 (2)	2.41 (3)	2.52 (2)	2.49 (2)	2.50 (2)	
SQ4.4: Confidence	2.18 (6)	2.25 (5)	2.27 (5)	2.17 (5)	2.21 (5)	
SQ4.5: CI	2.63 (1)	2.63 (1)	2.68 (1)	2.59 (1)	2.63 (1)	
SQ4.6: Ignore	2.32 (5)	1.96 (7)	2.21 (6)	2.11 (6)	2.16 (6)	
SQ4.7: Reproduce	2.05 (7)	2.14 (6)	2.07 (7)	2.11 (6)	2.09 (7)	
SQ4.8: Differentiate	1.70 (8)	1.85 (8)	1.76 (8)	1.77 (8)	1.76 (8)	
RQ3: Causes						Never / Rarely / Sometimes / Often
SQ5.1: Waiting	1.48 (3)	1.08 (5)	1.26 (4)	1.34 (4)	1.30 (4)	
SQ5.2: Concurrency	1.27 (5)	0.95 (7)	1.24 (5)	1.02 (5)	1.12 (5)	
SQ5.3: Setup/teardown	1.73 (1)	1.64 (1)	1.84 (1)	1.56 (1)	1.69 (1)	
SQ5.4: Resources	0.82 (7)	0.97 (6)	0.93 (7)	0.85 (7)	0.89 (7)	
SQ5.5: Network	1.63 (2)	1.21 (2)	1.53 (2)	1.36 (2)	1.44 (2)	
SQ5.6: Random	0.69 (8)	0.70 (9)	0.68 (9)	0.70 (8)	0.69 (9)	
SQ5.7: Time/date	1.01 (6)	1.12 (4)	1.11 (6)	1.02 (5)	1.06 (6)	
SQ5.8: Floating point	0.33 (10)	0.66 (10)	0.59 (10)	0.37 (10)	0.48 (10)	
SQ5.9: Unordered	0.69 (8)	0.78 (8)	0.84 (8)	0.63 (9)	0.73 (8)	
SQ5.10: Unknown	1.45 (4)	1.16 (3)	1.29 (3)	1.35 (3)	1.32 (3)	
RQ4: Actions						Never / Rarely / Sometimes / Often
SQ7.1: No action	1.40 (4)	0.91 (5)	1.41 (4)	1.01 (4)	1.19 (4)	
SQ7.2: Re-run	2.81 (1)	2.46 (2)	2.68 (1)	2.66 (1)	2.67 (1)	
SQ7.3: Document	1.58 (3)	1.67 (3)	1.59 (3)	1.65 (3)	1.62 (3)	
SQ7.4: Delete	0.86 (7)	1.06 (4)	1.09 (5)	0.80 (7)	0.94 (5)	
SQ7.5: Quarantine	0.74 (8)	0.79 (7)	0.81 (7)	0.73 (8)	0.77 (8)	
SQ7.6: Mark skip	0.98 (5)	0.85 (6)	1.04 (6)	0.84 (5)	0.93 (6)	
SQ7.7: Mark re-run	0.96 (6)	0.55 (8)	0.74 (8)	0.84 (5)	0.79 (7)	
SQ7.8: Repair	2.23 (2)	2.64 (1)	2.53 (2)	2.31 (2)	2.41 (2)	

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

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

After our thematic analysis for **SQ6**, we identified the following themes, in order of prevalence, regarding additional causes:
SQ6t1: External artifact: *An issue in an external service, library, or other artifact, that is outside the scope and control of the software under test.* As a potential cause of flaky tests, P8 reported "third-party artifacts, services, or dependencies ... which you do not have full control of ...". Responses of this prevalent theme were split between highlighting instabilities in remote services (in some instances a special case of **SQ5.5**), and issues in third-party libraries. The common aspect is that participants did not have control over the external artifact. On the external services side, **SQ11t7** is a special case of this theme, further evidencing its prevalence.
SQ6t2: Environmental differences: *Environmental differences between local development machines and remote build machines.* P21 referred to "environmental differences in local vs CI like different Java Virtual Machine (JVM) defaults." Almost all the responses in this theme made reference to CI. P97 offered a more nuanced explanation, highlighting how file system latency and concurrency-related issues may cause code to behave differently on a CI system. This theme is a special case of **SQ11t4** and directly supports **SQ2t1**.
SQ6t3: Host system issues: *Problems regarding the machines running the test suites.* In the words of P155, the most common aspect of this theme is "changes in hardware that the code and tests are running on". In many instances, this is a hardware analogue of **SQ6t2**, where a change in a machine is the cause of the flakiness.
SQ6t4: Test data issues: *Issues originating from the data used by test cases.* Some participants described flakiness caused specifically by test data. Most of these responses were brief and made reference to test data that was "deteriorated", "changing", or "external".
SQ6t5: Resource exhaustion: *Limited computational resources, such as memory and storage.* P49 wrote "unrelated system load on a shared resource causing low-level timeouts". P133 also made reference to system load from unrelated processes, giving antivirus software as an example. Other responses leaned more towards test cases that consume too many resources themselves. This theme is distinct from **SQ5.4**, which is specifically about mismanagement.
SQ6t6: OS differences: *Differences between operating systems (OS) or different versions of the same OS.* P62 described their experience after upgrading to a later version of Windows — "user interface (UI) changes with new OS. Things like EggPlant, that uses graphics. Moving to a new version of Windows (I think), changed the battleship gray ever so slightly and failed our UI tests". P76 explained how filesystem differences between OSes can cause flaky tests.
SQ6t7: Virtual machines: *Complications arising from the use of virtual machines or containers.* Put simply by P52, "the automation of virtual machines is asking for trouble". Vagrant and Docker were specific technologies referred to by the participants.
SQ6t8: UI testing: *Non-determinism inherent to the testing of UIs.* P147 wrote "UI not being in the expected state, i.e., keyboard not closed, or animations not completed when checking results". P100 specifically described how "random quirks in how Selenium works" caused them to have flaky tests. In many instances, this theme is a special case of **SQ5.1**, since UI test cases often have to wait for a specific element of a UI to be in the correct state [38].
SQ6t9: Conversion issues: *Inconsistencies when converting between data representations.* In the words of P48, "in my code that tests database interactions, I've run into issues where my coding language has more time precision than my storage ...". P103 relayed a similar experience regarding timestamps. While both participants referred to time, this theme is applicable to any data type.
SQ6t10: Timeouts: *Test execution exceeding a time limit and being prematurely terminated by the test runner.* P76 wrote "not waiting long enough for an environment to be set up". P79 referred to input and output operations occurring within a specific time limit.

We identified eight further themes regarding causes after analyzing the StackOverflow threads. In order of prevalence:
Ct1: UI Timing: *Test case does not wait for a user interface to be in the correct state.* This theme is a subset of **SQ5.1** regarding general asynchronicity and a special case of **SQ6t8** pertaining specifically to timing. This theme is related to **SQ6t3** since the execution speed of the machine is likely to significantly impact any timing issues.
Ct2: Logic error: *Error in the logic of the test code or the code under test.* This theme is broadly characterized by an oversight or misunderstanding on the part of the author of the test case. In one specific instance, a test case used an inappropriate method to wait for a condition in a UI [4]. This led to flakiness by **Ct1**, though since the root cause was that the developer misunderstood the use case of the waiting method, we placed this thread in **Ct2**.
Ct3: Shared state: *Test case depends on state shared with other test cases.* In one thread, the question describes a test case that shares a database connection with other test cases and only passes when executed in isolation [2]. This is an example of an OD test.
Ct4: Unknown: *The cause was never resolved.* Like its counterpart in the survey, **SQ5.10**, this theme was fairly prevalent.
Ct5: Setup/teardown: *Test case does not properly clean up after itself or fails to set up its necessary preconditions.* This is equivalent to **SQ5.3**. While OD tests were the main motivation for **SQ5.3**, the threads in **Ct5** describe test cases that do not appear to be OD. This theme was uncommon yet **SQ5.3** was rated as the most common by participants in the survey. This suggests that the most common causes are not necessarily the hardest for developers to repair.
Ct6: External library: *Bug in a third-party library or package.* This is a special case of **SQ6t1** pertaining specifically to third-party libraries. In one instance, an intermittent `NullPointerException` from an external package is the cause of flaky tests [1].
Ct7: Resource leak: *Test case does not release acquired resources.* This theme is a subset of **SQ5.4** specifically regarding test cases

that do not release resources, rather than general mismanagement. It is similar but not equivalent to **SQ6t5**, since the exhaustion of computational resources is not necessarily due to mismanagement.
Ct8: Improper mocking: *Test case does not mock an object or method correctly.* Like **Ct2**, this theme is unique to the StackOverflow analysis and describes flaky tests caused by improper mocking. A mock is a method or object that simulates the behavior of its real counterpart to make testing more straightforward [44].

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

RQ4: Actions. The bottom third of Table 2 presents the numerical analysis for **SQ7**. The most common action as scored by all participants was to simply re-run the failing build (**SQ7.2**). The second most common was to attempt to repair the flaky test (**SQ7.8**). After these two, there is a significant drop in mean score for the remaining actions. This implies that re-running the build and attempting to repair the flaky test are generally the most common actions developers take when encountering flaky tests.

The greatest difference in score between the participants who experienced flaky tests at least monthly and those who did not was for **SQ7.1**. This suggests that developers who experience flaky tests more often are more likely to take no action. There is also a considerable difference for **SQ7.8**, implying that developers who experience flaky tests less frequently are more likely to attempt to repair them. Furthermore, **SQ11t8** indicates that the costs of repair are too great for the potential gains. Arguably, this is less applicable when developers rarely experience flaky tests, which could partially explain the differences in **SQ7.1** and **SQ7.8**.

Following our thematic analysis for **SQ8**, we identified the following themes regarding actions, in order of prevalence:
SQ8t1: Emotive response: *An expression of anger or some other emotion.* This theme is generally equivalent to **SQ11t2**, but specifically in the context of a direct response to flaky tests.
SQ8t2: Alert proper person: *Inform other member or members of the development team about the flaky test.* In the words of P52, "tell the person who maintains that codebase". This theme is similar to **SQ7.3** but is more direct than just documenting the flaky test.
SQ8t3: Reorder tests: *Adjust the order of the test cases.* We placed two responses under this theme but they both had different motivations. P7 said "reorder tests to fail faster" and P111 said "reorder tests in case they are order-dependent". The former seems to be referring to test case prioritization [37]. The latter is talking about OD tests, but rather than repairing them they are seeking to find a test run order that does not manifest their flakiness [27].
SQ8t4: Repair resource: *Ensure a resource is in the correct state.* Summarized by P8, "when the test depends on the global state ... the test needs to be neither deleted/skipped, nor repaired, but rather, the state of the resource needs to be repaired ...". This theme is arguably a manifestation of **SQ11t5**, since the flaky test highlights a flaw in an aspect of the software beyond the test case code.
SQ8t5: Rewrite code under test: *Modify the code under test, as opposed to the test code.* P133 said "rewrite problematic code to make it more testable". This has clear links with **SQ2t2**, since it proves that a flaky test can highlight issues in the code under test. It is also a manifestation of **SQ11t5**, for the same reason as **SQ8t4**.

In order of prevalence, our analysis of the StackOverflow threads also resulted in the following themes pertaining to actions:
At1: Fix logic: *Repair a logic error.* All instances of this theme address an instance of **Ct2**. Given that **Ct2** is generally about API misuse, or inappropriate use of specific elements of an API, answers in **At1** typically highlight the correct API usage or recommend a more appropriate method for a particular purpose [4].
At2: Wait for condition: *Add an explicit wait for a condition.* Answers in this theme address most instances of **Ct1** and prescribe waiting for a specific condition, rather than a fixed time delay.
At3: Add mock: *Mock out an object or method.* This theme directly addresses **Ct8** but is also applicable to many other causes, such as **Ct1**. In one instance, the answer recommends mocking to address a timing issue that causes flakiness in a user interface test [5].
At4: Add/adjust external library: *Use a third-party library or adjust the version of a library already in use.* All answers in this theme address an instance of **Ct6**. In one specific thread, the answer highlights the latest version of a particular third-party library that previously contained a bug that was causing the flakiness [1].
At5: Fix setup/teardown: *Repair insufficient setup or teardown procedure.* This theme directly addresses **Ct5** by suggesting changes to setup and/or teardown methods that were causing flakiness.
At6: Isolate state: *Remove dependency on a shared state.* This theme mostly addresses **Ct4**, but not always. One accepted answer suggests decoupling shared database connections [2].

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

Table 3: Our recommendations, supported by our results and previous literature. Targets researchers (🎓) and developers (</>).

		Supported by	
	Recommendation	**Results**	**Literature**
</>	**R1: Consider beyond code**. The definition of a flaky test should include factors beyond the test case code or the code under test, such as properties of the execution environment. Developers should consider the behavior of test cases in different environments, particularly when going from a local environment to CI.	**SQ2t1**, **SQ6t2**, **SQ11t4**	[22, 42, 49]
</>	**R2: Not completely useless**. Flaky tests may indicate a flaw in the code under test or another aspect of the software system. Therefore, developers should not write them off as completely useless.	**SQ2t2**, **SQ8t4**, **SQ8t5**, **SQ11t5**	[17]
🎓	**R3: Impact on CI**. Flaky tests can become an obstacle to the effective deployment of CI. Researchers should consider the creation and evaluation of new approaches to better mitigate this trend.	**SQ4.5**	[14, 23]
</>	**R4: Careful setup/teardown**. Insufficient setup and teardown is a common cause of flaky tests. Developers should exercise particular care when writing setup and teardown methods for their test suites.	**SQ5.3**, **Ct5**, **At5**	[9]
🎓	**R5: Identify root causes**. It is difficult to manually determine the root cause of many flaky tests. Researchers should continue to develop automated techniques for this challenging task [24].	**SQ5.10**, **Ct4**	[30]
</>	**R6: Repair promptly**. The results suggest that participants who said they experienced flaky on at least a monthly basis may be more likely to ignore genuine test failures, more likely to take no action in response to flaky tests, and less likely to attempt to repair them. Therefore, developers should to repair flaky tests as soon as possible after identifying them to avoid them accumulating and potentially being ignored.	**SQ4.6**, **SQ7.1**, **SQ7.8**, **SQ11t8**	[19]

4 RECOMMENDATIONS

Table 3 lists our six recommendations. We found that **SQ2t1**, the most common theme in **SQ2**, extends our proposed definition of flaky tests to consider factors beyond the code of the test case and the code under test, particularly the execution environment. Furthermore, **SQ6t2** represents environmental differences as a cause of flaky tests and is a special case of **SQ11t4**. This supports a line of research that considers how changes in the implementations of third-party libraries can manifest flaky tests [22, 42, 49]. These results and previous studies are the basis for **R1**.

The second most common theme in **SQ2**, **SQ2t2**, represents the idea that flaky tests can indicate that the code under test is flawed. Following our thematic analysis for **SQ8**, we identified **SQ8t4** and **SQ8t5**, regarding repairing a resource and the code under test respectively, as actions in response to flaky tests. Furthermore, **SQ11t5** relays the concept that flaky tests have utility. These results form the foundation of **R2**, along with one of the findings of Eck et al. [17], that, for certain types of flaky test, developers sometimes considered the cause to originate from the code under test.

For **SQ4**, we found that participants strongly agreed that flaky tests hinder CI. This is the motivation for **R3**, along with the findings of Hilton et al. [23], who asked developers to estimate the number of weekly failing CI builds caused by genuine and flaky test failures. They found no significant difference between the two estimates. This also supports Durieux et al. [14], who found that 47% of previously failing CI builds that were manually restarted passed without changes, suggesting the influence of flaky tests.

Previous studies identified waiting for asynchronous events (**SQ5.1**) and concurrency (**SQ5.2**) to be the most common causes of flaky tests [17, 30, 40]. According to our survey, these causes were only the fourth and fifth most common. We found inadequate setup/teardown (**SQ5.3**) to be the most common and also identified this theme in our StackOverflow analysis (**Ct5** and **At5**), forming the basis of **R4**. However, these studies were based on previously repaired flaky tests, whereas in our case, the participants reported the frequency of each cause according to their experience. It could be that the most common causes of flaky tests are not necessarily the ones that developers prioritize for repair. On the other hand, Bell et al. [9] suggested that the difficulty of writing correct setup and teardown could cause OD tests. However, as attested by **Ct5**, this cause may not always result in a flaky test that is OD.

For **SQ5**, we found that participants scored **SQ5.10**, regarding unknown causes, as the third highest overall. Similarly, we found that the cause of the flakiness was never resolved in many of the StackOverflow threads (**Ct4**). This is reflected by Luo et al. [30], who categorized the causes of the repaired flakiness in 201 commits and found "Hard to classify" to be the second most common category. Taken together, these results are our rationale for **R5**.

We found that participants who said they experience flaky tests on at least a monthly basis scored **SQ4.6**, regarding ignoring potentially genuine test failures, considerably higher than those who did not. Martin Fowler [19] wrote that flaky tests have an "infectious" quality, and as they proliferate, developers may ignore test failures in general. Furthermore, our results for **SQ7.1** and **SQ7.8** indicate that developers who experience more flaky tests may be more likely to take no action against them and less likely to attempt to repair them. We therefore suggest **R6** in response to these findings.

5 RELATED WORK

Luo et al. [30] performed an empirical study that investigated the causes, manifestations, and fixing strategies of flaky tests. As objects of analysis, they used 201 commits that repaired flaky tests in a range of Apache Software Foundation projects. They introduced ten cause categories that have since been used in subsequent research [17, 21, 25, 40]. They found that the top three causes of flaky tests were asynchronicity, concurrency, and test-order dependency (OD tests). Unlike our study, Luo et al. did not base any of their findings on the self-reported experiences of developers. In that respect, their methodology is closer to our StackOverflow analysis.

Eck et al. [17] performed a related study. They asked 21 developers from Mozilla to classify 200 flaky tests that they had previously repaired and also conducted a broader online survey, receiving

121 responses. Using Luo et al.'s ten categories as a starting point, through their Mozilla study, they identified four additional causes, including overly restrictive assertion ranges and platform dependency (broadly similar to **SQ6t2** and **SQ6t6**, respectively). To keep our results as general as possible, we chose not to focus on any particular organization in any part of our study. Moreover, we included additional objects of analysis beyond developers' testimonies to limit any self-reporting bias [13]. As part of their broader survey, they asked developers to estimate how often they dealt with flaky tests. Their results are very similar to ours for **SQ3**.

As part of a wider study on the uptake of CI, Hilton et al. [23] deployed a survey at Pivotal Software. Among other questions, the survey asked developers to estimate the number of CI builds failing each week due to genuine test failures and due to flaky test failures. Following a Pearson's chi-squared test, they found no statistically significant difference between the genuine and the flaky distributions. Our findings confirm that flaky tests are very prevalent (**SQ3**) and that flaky tests are a hindrance to CI (**SQ4.5**).

Gruber et al. [20] also deployed a survey about flaky tests, with a specific focus on the support that developers need from tools.

6 CONCLUSIONS AND FUTURE WORK

We deployed an online survey about flaky tests, not restricted to any organization, and received 170 responses. It focused on understanding how developers define and react to flaky tests and their experiences of the causes and impacts. We also procured a dataset of 38 StackOverflow threads, upon which we performed thematic analysis to identify further causes and repair strategies. Ultimately, we offer six actionable recommendations for both researchers and developers. As part of future work, we plan to conduct a larger-scale study with a greater volume of participants to improve the generalizability of our findings. We also intend to include focused surveys and interviews of developers from a variety of organizations.

REFERENCES

[1] 2014. *NPE Inside Robotium.* https://stackoverflow.com/questions/23519395/npe-inside-robotiumk
[2] 2016. *Flaky Tests with DatabaseCleaner and Transactions. How to Debug?* https://stackoverflow.com/questions/37560303/flaky-tests-with-databasecleaner-and-transactions-how-to-debug
[3] 2016. *How Do You Label Flaky Tests Using JUnit?* https://stackoverflow.com/questions/39538400/how-do-you-label-flaky-tests-using-junit
[4] 2017. *Can I Detect If an Element (Button) Is "Clickable" In My RSpecs?* https://stackoverflow.com/questions/48027118/can-i-detect-if-an-element-button-is-clickable-in-my-rspecs
[5] 2019. *Detox: Detect That Element was Displayed.* https://stackoverflow.com/questions/59412749/detox-detect-that-element-was-displayed
[6] 2021. *Replication Package.* https://github.com/flake-it/flaky-test-survey-replication-package
[7] 2021. *Sheffield Digital.* https://sheffield.digital/
[8] 2021. *StackOverflow Developer Survey.* https://insights.stackoverflow.com/survey/2021
[9] J. Bell and G. Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proc. ICSE.*
[10] Box. 2021. *Flaky.* https://github.com/box/flaky
[11] D. S. Cruzes and T. Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *Proc. ESEM.*
[12] Software Engineering Daily. 2019. *Facebook Engineering Process with Kent Beck.* Retrieved 9/01/2021 from https://softwareengineeringdaily.com/2019/08/28/facebook-engineering-process-with-kent-beck/
[13] S. Donaldson and E. Grant-Vallone. 2002. Understanding Self-Report Bias in Organizational Behavior Research. *Journal of Business and Psychology* 17, 2 (2002).
[14] T. Durieux, C. L. Goues, M. Hilton, and R. Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proc. MSR.*
[15] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and Misailovic S. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proc. ISSTA.*
[16] S. Dutta, A. Shi, and Misailovic S. 2021. FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds. In *Proc. ESEC/FSE.*
[17] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proc. ESEC/FSE.*
[18] L. Eloussi. 2016. *Flaky Tests (And How to Avoid Them).* https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60
[19] M. Fowler. 2011. *Eradicating Non-Determinism in Tests.* https://martinfowler.com/articles/nonDeterminism.html
[20] M. Gruber, , and G. Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need to Address It. In *Proc. ICST.*
[21] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proc. ICST.*
[22] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. 2015. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specification. In *Proc. FSE.*
[23] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proc. FSE.*
[24] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proc. ISSTA.*
[25] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *Proc. ICSE.*
[26] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *Proc. ICST.*
[27] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *Proc. ISSTA.*
[28] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *Proc. ISSRE.*
[29] B. Lee. 2019. *We Have a Flaky Test Problem.* https://medium.com/scopedev/how-can-we-peacefully-co-exist-with-flaky-tests-3c8f94fba166
[30] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proc. FSE.*
[31] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. 2019. Predictive Test Selection. In *Proc. ICSE-SEIP.*
[32] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-Scale Continuous Testing. In *Proc. ICSE-SEIP.*
[33] J. Micco. 2016. *Flaky Tests at Google and How We Mitigate Them.* https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[34] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. 2012. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *Proc. ICSM.*
[35] M. Nejadgholi and J. Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *Proc. ASE.*
[36] Q. Peng, A. Shi, and L. Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proc. ISSTA.*
[37] J. A. Prado Lima and S. R. Vergilio. 2020. Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study. *IST* (2020).
[38] K. Presler-Marshall, E. Horton, S. Heckman, and K. T. Stolee. 2019. Wait Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness. In *Proc. AST.*
[39] M. T. Rahman and P. C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated With Firefox Builds. In *Proc. ESEC/FSE.*
[40] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. In *Proc. ICSE.*
[41] A. Shi, J. Bell, and D. Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proc. ISSTA.*
[42] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-Deterministic Specifications. In *Proc. ICST.*
[43] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *Proc. ESEC/FSE.*
[44] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. 2017. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *Proc. MSR.*
[45] P. Sudarshan. 2012. *No More Flaky Tests on the Go Team.* https://www.thoughtworks.com/en-gb/insights/blog/no-more-flaky-tests-go-team
[46] S. Thorve, C. Sreshtha, and N. Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *Proc. ICSME.*
[47] A. Vahabzadeh, A. A. Fard, and A. Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *Proc. ICSME.*
[48] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. In *Proc. ESEC/FSE.*
[49] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi. 2021. Domain-Specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications. In *Proc. ICSE.*
[50] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proc. ISSTA.*