Received: 15 April 2021 | Revised: 2 August 2021 | Accepted: 5 August 2021

DOI: 10.1002/stvr.1791

Empirical analysis of practitioners' perceptions of test flakiness factors

Azeem Ahmad | **Ola Leifler** | **Kristian Sandahl**

Linköping University, Linköping, Sweden

Correspondence
Azeem Ahmad, Department of Computer Science, Linköping University,
581 83 Linköping, Sweden.
Email: azeem.ahmad@liu.se

Funding information
Chalmers Tekniska Högskola; Linköpings Universitet

Summary

Identifying the root causes of test flakiness is one of the challenges faced by practitioners during software testing. In other words, the testing of the software is hampered by test flakiness. Since the research about test flakiness in large-scale software engineering is scarce, the need for an empirical case-study where we can build a common and grounded understanding of the problem as well as relevant remedies that can later be evaluated in a large-scale context is a necessity. This study reports the findings from a multiple-case study. The authors conducted an online survey to investigate and catalogue the root causes of test flakiness and mitigation strategies. We attempted to understand how practitioners perceive test flakiness in closed-source development, such as how they define test flakiness and what practitioners perceive can affect test flakiness. The perceptions of practitioners were compared with the available literature. We investigated whether practitioners' perceptions are reflected in the test artefacts such as what is the relationship between the perceived factors and properties of test artefacts. This study reported 19 factors that are perceived by professionals to affect test flakiness. These perceived factors are categorized as *test code*, *system under test*, *CI/test infrastructure*, and *organization-related*. The authors concluded that some of the perceived factors in test flakiness in closed-source development are directly related to non-determinism, whereas other perceived factors concern different aspects, for example, lack of good properties of a test case, deviations from the established processes, and ad hoc decisions. Given a data set from investigated cases, the authors concluded that two of the perceived factors (i.e., test case size and test case simplicity) have a strong effect on test flakiness.

KEYWORDS

flaky tests, non-deterministic tests, practitioners' perceptions, software testing, test smells

1 | INTRODUCTION

Regression testing, automatic or manual, is intended to ensure that changes made in any part of the system do not break existing functionality. Developers submit code changes with the expectation that test failures will be associated with the code modifications. Unfortunately, rather than being the result of changes to the code, some test failures occur due to flaky tests. In the literature, the most common definition of a flaky test is: a test that exhibits both passing and failing outcomes when no changes are introduced into the code base [1]. King et al. extend this definition [2]: *"flaky tests exhibit both passing and failing results when neither the code nor test has changed"*. Flaky tests are defined as *"unreliable tests whose outcome is not deterministic."*

Many large-scale software projects suffer from a high amount of test flakiness. For example, Google has reported that 80% of failing tests were due to flakes and only about 20% were actual regressions [3]. Another study confirmed 1 in 7 of the tests at Google sometimes fail due to test flakiness [4]. Labuschagne et al. studied 61 Java projects to examine test failures. They concluded that 13% of these failures were due to flaky tests [5]. Hilton et al. [6] reported that half of all builds failed due to tests failures containing flakiness. Flaky tests are common in large code bases, and the current approaches to handling flakiness are not satisfactory [7]. We have observed that current studies for addressing test flakiness mostly depend on software artefacts (i.e., source code and test cases), which are just a small part of the bigger problem, and many core issues are still not addressed for test non-determinism. In this study, we therefore take a step back to investigate the perceptions of test flakiness among practitioners in a closed-source development. We believe that practitioners at software companies are struggling to understand the root causes of test flakiness and how to avoid or reduce it, thus forming a local and narrow opinions about test flakiness. Practitioners sometimes favour local opinion over empirical evidence when adopting new techniques, which makes perception important [8]. The need to understand practitioners' perceptions with respect to software engineering has received significant attention from the scientific community. Researchers have investigated practitioners' perceptions of continuous integration [9], software design (e.g., code smells or exception handling) [10-13], software testing [14-16], and software quality [17-20] and have observed differences in practitioners' perceptions of the subjects under investigation. Since flaky tests are a serious concern of software professionals, we decided to investigate professionals' perceptions about what factors are perceived to affect test flakiness. Capturing these perceptions together with the comparison of prior work will enhance the understanding of test flakiness to an extent where it is non-ambiguous resulting in avoiding problems of people misunderstanding each other. A similar approach was adopted by Eck et al. [21] in a study where they investigated developers' perceptions of the causes, fixing efforts, significance, and challenges of test flakiness in an open-source project (Mozilla).

In addition to understanding the perceptions among practitioners, we investigated whether or not the developers' perceptions match with what they have marked as flaky or not. The idea is to investigate what practitioners perceive and whether these perceptions are reflected in the test artefacts. Only those factors that relate to properties of test code are assessed in this way such as test case simplicity and size. For example, do flaky tests contain on average more assertions than non-flaky tests? Are flaky tests bigger (i.e., lines of code)? The other factors such as *perseverance of a team to detect and prevent flakiness* will not be related to properties of test code, thus were not included in the investigation.

Several studies have investigated the relationship between test smells and test flakiness [7,22] but did not conclusively state that the test flakiness is only caused by the test smells. We inspected test artefacts at two companies and listed test smells that induce test flakiness together with corresponding mitigation strategies. We investigated the following research questions in this study:

> *RQ1: What are the root causes of test flakiness in closed source industry and how do professionals address test flakiness?*
> *RQ2: What factors do practitioners perceive as affecting the test flakiness?*
> *RQ3: Can perceived factors (i.e., test case size and simplicity) explain whether a test case is flaky or not?*

The major contributions of this paper are as follows.

C1: The survey results about what are the root causes in test flakiness and how practitioners in closed-source industry manage test flakiness. (RQ1, Section 4.1)

C2: A list of identified test smells that are known to induce test flakiness in test artefacts as well as the mitigation strategies to address those test smells. (RQ1, Section 4.2)

C3: An attempt to study practitioners' perceptions of test flakiness, such as how they define test flakiness and what factors (other than test smells), they believed, can either increase or decrease test flakiness. These perceived factors were further mapped to current literature to categorize them. (RQ2, Section 4.3)

C4: A comparison of our findings with what was presented by Eck et al. [21]. During our study, we identified 8 new factors that have not been mentioned in Eck et al. [21]. (RQ2, Section 4.3)

C5: We investigated whether or not the developers' perceptions match with what they have actually marked as flaky or not. (RQ3, Section 4.4). Only those factors that relate to the properties of test code are assessed in this way such as test case simplicity and size. The other factors such as *perseverance of a team to detect and prevent flakiness* will not be related to properties of test code, thus were not included in the investigation.

Related work is presented in Section 2. The research protocol is presented in Section 3. Section 4presents results such as answers to RQ1, RQ2, and RQ3 in Sections 4.1–4.4, respectively. Section 5presents an evaluations of the perceived factors. A discussion is presented in Section 6 and potential threats to our study's validity in Section 7. The conclusion is presented in Section 8.

2 | RELATED WORK

Luo et al. [7] categorized the causes of test case flakiness by investigating 52 open-source projects. They manually inspected 201 commits of selected projects. Asynchronous wait (45%), concurrency (20%), and test-order dependency (12%) were found to be the most common causes of test flakiness. Based on these findings, Luo et al. suggest avoiding specific code smells that lead to test flakiness. Another empirical study of the root causes of test flakiness in Android Apps was conducted by Thorve et al. [22] by analysing the commits of 51 Apache open-source projects. Thorve et al. complemented the results of Luo et al. and Palomba and Zaidman, but they also reported two additional test smells categorized as user interface and program logic that induce test flakiness in Android Apps. In addition to these studies, researchers have investigated the root causes of test flakiness in different applications such as web applications [23,24], Android apps [25], and Automated REST APIs [26].

Eck et al. [21] investigated developers' perceptions of the causes, fixing efforts, significance, and challenges of test flakiness, in open-source project. Their work discovered four new types of causes such as (1) too restrictive range in test assertions (i.e., valid output values are outside the assertion range), (2) test case timeout, (3) platform dependency, and (4) test suite timeout. After a survey and interview with developers, they concluded that the broader view of test flakiness should include both source code and its management as well as the team's organization, resolution process, and test case design. We designed this study to capture more factors covering more areas of software engineering than presented by [21] (a comparison of their work with our findings has been provided in different parts of this paper, wherever needed, particularly in Table 6).

In addition to identifications of test smells inducing test flakiness, researchers have proposed tools and techniques to detect test flakiness. Bell et al. proposed a new technique called *DeFlaker*, which monitors the latest code coverage and marks the test case as flaky if the test case does not execute any of the changes [27]. Another technique called *PRADET* [28] does not detect flaky tests directly, rather it uses a systematic process to detect problematic test order dependencies. These test order dependencies can lead to test non-determinism. Dutta et al. categorized the common root causes of test flakiness in applications that were built on machine learning frameworks such as Pyro, PyMC3, TensorFlow-Probability and PyTorch [29]. They developed a technique called *FLASH* which detects flaky tests due to assertions passing and failing during different executions while maintaining a same codebase. The *FLASH* technique is restricted to only those assertions that failed due to differences in the sequence of random numbers of the same test in different executions. Lam et al. developed a framework named *RootFinder* that instruments flaky tests to achieve logs of various runtime properties [30]. Later, this tool finds differences in the logs of passing and failing runs [30]. In another study by Lam et al., an automated solution named *Flakiness & Time Balancer* is presented to reduce the test failures caused by synchronous calls. Shi et al. [31] studied the effects of flaky tests on mutation testing. They concluded that due to test flakiness, many mutants are not covered during testing, although the prior coverage collection indicated that it should be [31].

King et al. presented an approach that leverages Bayesian networks for flaky test classification and prediction [2]. This approach considers flakiness as a disease that can be mitigated by analysing the symptoms and possible causes. This approach identifies, quarantines, and fixes flaky tests in continuous integration pipelines. This approach was adopted by 6 teams within a software company and improved the stability of CI pipeline (i.e., reducing flaky tests) by as much as 60%. Pinto et al. evaluated the performance of different machine learning classifiers such as Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour on 25 open-source projects to predict flaky tests as well as find the vocabulary of flaky tests. They concluded that Random Forest performed best as compared to other classifiers. Moreover, they concluded that projects with high involvement of IO exhibit more test flakiness. Twenty words that frequently appeared in flaky tests were identified in the study. The top five words were job, table, id, action, and oozie. These words were associated with executing tasks remotely or using an event queue.

We take one step further in a different direction to (1) detailed quantitative analysis of the survey on issues and mitigation strategies, (2) capture practitioners' perceptions in the closed-source development, (3) comparative analysis of test flakiness within the investigated companies and with what Eck et al. presented in their study, and (4) identification of the test smells that are known to induce test flakiness in test cases collected from different cases.

TABLE 1 Information about case companies

Case	Business	Total empl.	Designations	Participant's experience in test flakiness	Test flakiness in CI[a]	Automation level[b]
A	Surveillance	5k	Testers, Managers, Developers	Yes	None	High
B	Water	5k	Testers, Managers, Developers	Yes	Low	High
C	Medical	3k	Testers, Developers	Yes	High	Moderate
D	Automotive	25k	Testers, Managers, Developers	Yes	High	High
E	Automotive	1k	Testers, Developers	Yes	High	None

[a] Low: Flaky tests appearing occasionally for example, during day-time savings, modifying CI infrastructure, unexpected circumstances, and so forth. High: Flaky tests appearing frequently after executing each build or every other build.
[b] High: The CI pipeline support automation from commits to deployment with none or very little human intervention. Moderate: The CI pipeline support automation from commit to testing and human intervention is required for deployment.

TABLE 2 Data collection and validation protocol

Case	Type	Duration (Min)	Date	Participants designation
A, B, C, D, E	Survey^{DC}	-	2018-10	10^{T}, 8^{D}
A, B, D, E	Workshop^{DC}	120	2018-11	6^{T}, 4^{D}
C	Workshop^{DC}	120	2018-12	2^{T}, 2^{D}
A	Interview^{DC},Site Visit	180	2018-12	2^{T}
A, B, C, D, E	Workshop^{DV}	120	2019-03	6^{T}, 4^{D}

Abbreviations: *D*, developer; *DC*, data collection; *DV*, data validation; *T*, test lead.

3 | RESEARCH METHODOLOGY

3.1 | Case descriptions

Table 1 provides detailed information about case companies; (1) what type of businesses the companies are involved in, (2) the number of employees, (3) selected teams for investigation in this study, (4) participants' experience in handling test flakiness, (5) how much test flakiness companies have observed during continuous integration, and (6) what level of test automation the companies have achieved. We selected companies that have different degrees of test flakiness to cover a broad spectrum of the participants.

3.2 | Data collection

In all five cases, we collected data through online and physical workshops, site visits, and semi-structured interviews. Most of the interactions were conducted online through Zoom or Skype. We recorded the conversation and took notes during all workshops with prior permission from the participants. Table 2 presents detailed information about data collection and validation protocol with respect to the number of participants, visit dates, and so forth.

3.2.1 | Survey

Prior to study, we conducted an initial survey to assess the case companies' interest in test flakiness through a Google form (step 1 in Figure 1). The survey can be accessed online.[1] The survey contains the necessary definitions of terms (i.e., root causes of test flakiness) and terminologies. The survey contains multiple choice questions and prepared by analysing prior literature in test flakiness. The survey was answered by 18 participants from 5 different companies. The survey provided answers to *RQ1*.

[1] https://tinyurl.com/yxpw3vgu

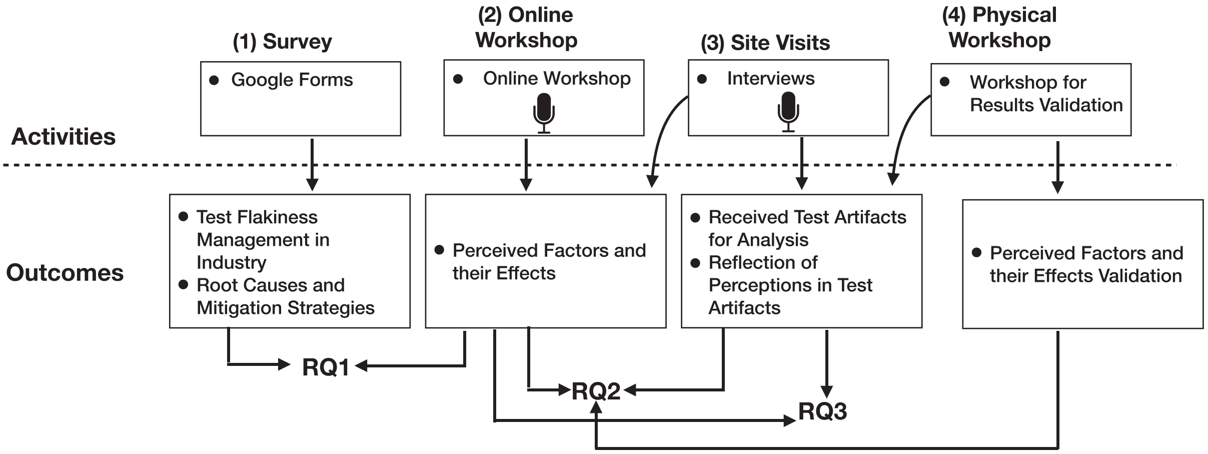
(1) Survey
Google Forms
(2) Online Workshop
Online Workshop
(3) Site Visits
Interviews
(4) Physical Workshop
Workshop for Results Validation
Activities
Outcomes
Test Flakiness Management in Industry
Root Causes and Mitigation Strategies
Perceived Factors and their Effects
Received Test Artifacts for Analysis
Reflection of Perceptions in Test Artifacts
Perceived Factors and their Effects Validation
RQ1
RQ2
RQ3

FIGURE 1 Research protocol for data collection, analysis, and validation

3.2.2 | Online workshops

After the online survey, we analysed the results and prepared open-ended questions for the online workshops (step 2 in Figure 1). The set of questions, that were discussed during the workshop, can be accessed online.[2] Participants did not have access to the set of questions before the workshop. We conducted a 2-h long workshop to collect data from cases A, B, D, and E. This workshop was conducted online through Zoom, and 10 participants (6 testers and 4 developers) from four companies participated. A similar 2-h online workshop was conducted with case C on a different occasion due to fact that the participants in case C were not available during first workshop. We encouraged the discussion among the participants during the workshop but we did not expect them to reach any consensus. We analysed individual answers in this study.

3.2.3 | Site visit

Later, one of the authors visited company A to conduct an interview with 2 testers (step 3 in Figure 1). The interview was 180 min long and was recorded and transcribed into an Excel sheet for analysis. The reason for visiting only one company was a desire to look at their testing process and documentation in person, due to their claim of having no flaky tests. The idea was to extract tacit knowledge from the company's day-to-day practices. We received a complete test suite consisting of 1609 test cases from case A as part of the data collection (during step 3). In addition, the authors received 30 tests marked as flaky and 120 marked as non-flaky from case B. An automated script was executed to identify the test smells that induce test flakiness, from the study [7,21], which were present in the test code. Whereas in the case B, two of the authors manually investigated flaky test cases code to investigated whether test smells that induce test flakiness were present in the test cases.

To investigate what practitioners perceive and whether these perceptions are reflected in the test artefacts, we ran an automated script to find non-commented lines of code and the number of assertions in each test case.

3.3 | Preparation

During data collection, two authors took notes in addition to an audio recording. Data were anonymized in such a way that it could not be traced back to the individual participants. We obtained informed consent from the participants for an audio recording. The audio recording was transcribed word-by-word using an audio player. The transcription was saved into the Excel sheet. Each cell in the Excel sheet contains texts from one person during a conversation, further labelled with the speaker. The names of participants were also anonymized with `PX`, where `X` is the number assigned by us to the participant for increasing the readability and traceability in within the text. Two of the authors checked the transcriptions to find any discrepancies or missing information.

[2] https://tinyurl.com/y3mvvypo

3.4 | Data analysis

We coded all the transcripts from the workshop and interviews according to *open coding* [32]. Each paragraph of the transcripts was labelled with one or more codes. Later, we compared all paragraphs from different companies to collect similar codes (axial codes). These codes were further divided as the influencing factor and their effect such as whether the factors were believed either to increase or decrease test flakiness. Table 3 presents an example of how we extracted the perceived factors and their effects on flakiness. We have used two different symbols to denote the effects of the factors in test flakiness. A plus sign (+) represents an increase and minus sign (−) represents a decrease of test flakiness. We have provided all of the quotes, in this paper, from participants as spoken during the interview or workshop, that is, untranslated transcriptions of what they said in English. All our subjects have experience in flaky test detection and removal.

3.5 | Data validation

After analysis, the initial codes were checked by two of the researchers. Later, we conducted a physical workshop with ten different participants from five companies to validate the results. For the data validation exercise, we selected different participants from the ones who had participated in the earlier data collection phase to avoid biases in the results. This workshop was conducted for 120 min and participants were provided with the list of influencing factors together with their descriptions. Participants were asked to rank the importance of the factors based on the Likert scale (i.e., 1 to 5, *Strongly Agree* to *Strongly Disagree*). Each factor was discussed with the participants to ensure that they understood it. In addition to the ranking of factor's importance, participants were asked to rank the effect (see Section 3.4) of the factors on test flakiness based on a similar Likert scale.

4 | RESULTS

We presented the results the same way we collected the data and investigated research questions. First, we present the survey results following the perceived factors and their effect on test flakiness. Later we report the test smells found in the test cases.

4.1 | Survey results—RQ1

As shown in Figure 2a, all participants from the five companies shared that they have observed test flakiness during integration testing—testing collections of modules which have been integrated into subsystems. Eleven participants observed it during system testing—subsystems are integrated to make up the entire system. Subsystem testing is concerned with finding errors that result from unanticipated interactions within subsystems or system components. Only one participant observed test flakiness in unit testing. Flakiness is easily captured during unit testing due to the fact that unit testing deals with atomic methods.

All investigated companies have test code review processes (Figure 2b), but flaky tests can still sneak into the test suite. All participants reported that their test design processes do not include guidelines to prevent flakiness. We observed that another reason for having test flakiness could be that practitioners do not label flaky tests as represented by Figure 2g. All participants except one answered "No" to the question *"If you notice a test case as flaky, do you use any type of annotation such as '@FlakyTest', '@Repeat', '@Ignore' or '@ReRunThis'."* The only person who answered "Yes" to this question belonged to company A. In case there are only a few flaky tests, the reason for not labelling the test case is justified but as represented in Figure 2c, 13 out of 18 participants estimated that flaky tests accounted for

TABLE 3 Identification of influencing factors and their effect

Original text	"for the project we have now, where we lack requirements, the reason for test flakiness was test case code. Some of the changes were made in a test case. This is for an undefined project and unclear requirement"
Identified factor	Requirements clarity
Effect	- Test flakiness (decrease test flakiness)
Textual description	Clear requirements decrease test flakiness
Notation	Requirements clarity → - Test flakiness

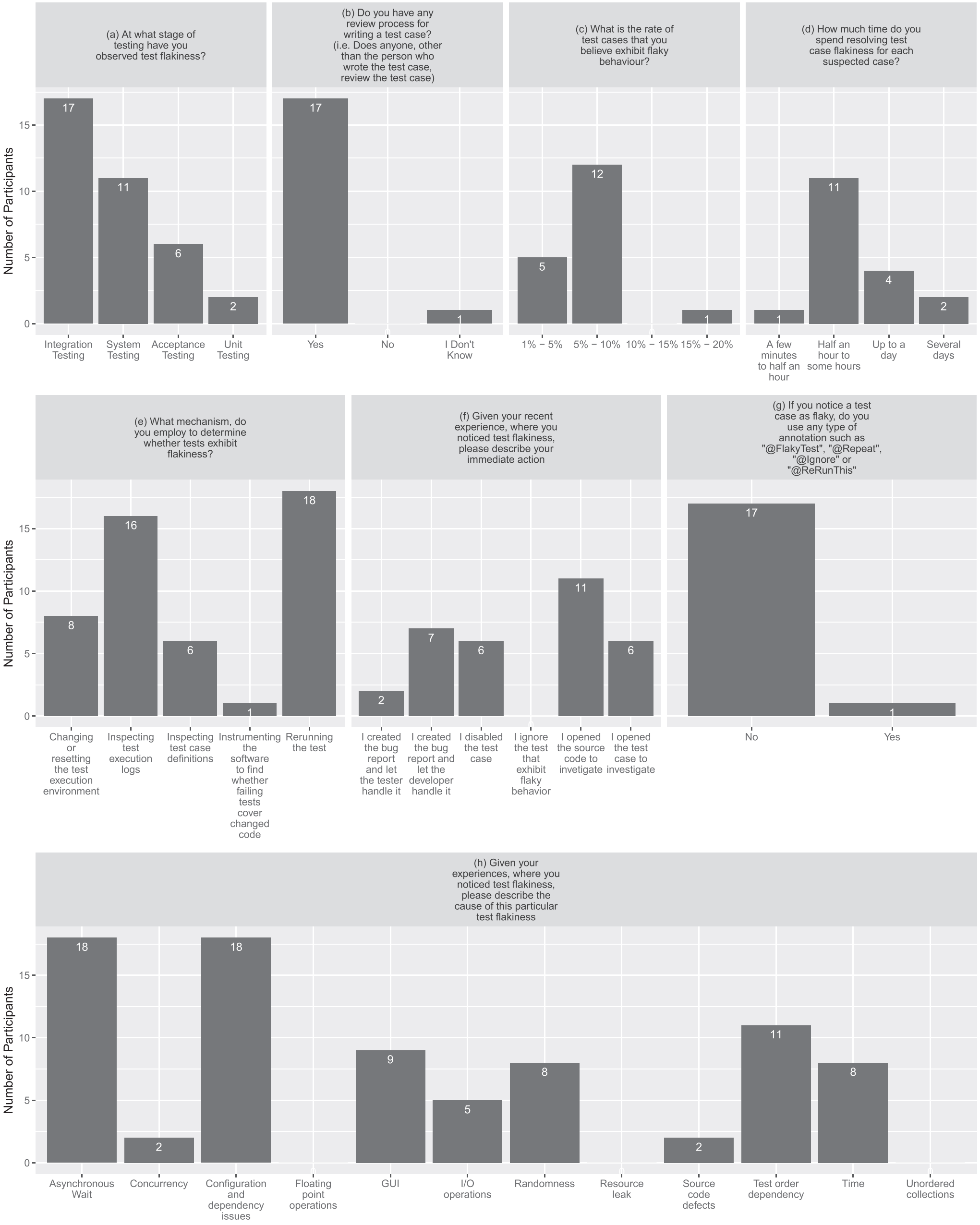

FIGURE 2 Survey results presented with questions in each facet

5%–10% of their total tests, while 4 other participants reported 1%–5% flaky tests. These percentages may appear small, but if the companies have many thousands of test cases, then these percentages matter. Our results can be compared with the study of Mozilla [21] where 100–150 flaky tests were detected, every week [21].

More than 50% of the participants shared that they spent between half an hour and an entire day resolving test case flakiness (Figure 2d). Four participants shared that they have spent up to a day to resolve test flakiness on many occasions. At the other extreme, only one participant shared that he spent a few minutes to half an hour, only on a few occasions, to resolve issues related to test flakiness. The results of our analysis (closed-source) overlap significantly with Eck et al [21]. The fixing efforts (very easy to very hard) described by developers in [21] range from 1.0 to 4.0 given the nature of test flakiness and matches to what we have identified that the fixing effort ranges from half an hour to several days. This time does not include administrative hours such as creating bug reports or assigning issues to concerned user; however, it does include time to inspect test execution logs, test case definitions, or resetting the testing environment, etc.

The most common mechanism to determine whether tests exhibit test flakiness is to rerun the tests. This is similar between closed-source and open-source industry. (Figure 2e). Figure 2e lists *"Inspecting the test execution logs"* as the second most common approach followed by *"Changing or resetting the test execution environment."* Only one participant reported that the company instrumented the software to find whether failing tests cover changed code. We assumed that this variance in activities to detect test flakiness is one of the reasons that developers spend a lot of time to resolve test flakiness.

We observed high variance in participants' response of the question *"Given your recent experience, where you noticed test flakiness, please describe your immediate action."* Practitioners performed different actions such as investigated source code or test case code, created a bug report and let other professionals handle it or disabled the flaky test.

In addition to capturing the different aspects of managing/identifying test flakiness, we investigated the root causes of test flakiness in terms of the test smells experienced by the professionals. Many studies [7,21,22,33] have investigated the relationship between test smells and test flakiness. We provided the list of test smells in the survey that is known to relate to test flakiness. "Asynchronous wait" and "configuration & dependency issues" have been mentioned as a major root cause of test flakiness by all participants, as represented by Figure 2h. These are the main root causes because companies have different types of hardware, running with a different platform (Windows, OSX, and Linux with different kernels) and libraries being used (open-source or commercial). The other major concerns by approximately more than half of the participants were "GUI," "I/O operations," "Randomness," "Test order dependency," and "Time." As represented in Figure 2h, four of the test smells such as "Concurrency," "Resource leak," "Floating-point operations," and "Unordered collections" were not experienced by any of the participants.

4.2 | Root causes in terms of test smells and mitigation strategies—RQ1

Table 4 presents information about test smells in the test cases received from companies. Case A shared their complete test suite containing 1609 test cases. Since case A did not mark which test cases are flaky, we ran an automated script to identity test smells that are known to induce test flakiness and reported by many researchers in [7,21,33]. We identified 402 (22%) test cases that contain test smells and may exhibit flaky behaviour. After the identification of test smells, we assigned it to predefined categories by researchers in [7,21,33]. We received 150 test cases from case B where 30 test cases were marked as flaky. These test cases were marked as flaky and include the information of causes (i.e., test smells) and mitigation strategies.

4.2.1 | Async wait

A test makes an asynchronous call and does not wait properly for the result of the call to become available leading to test flakiness. In the context of case B, during several occasions, the test case does not receive acknowledgment when it sends the reset command to the device under test leading to nondeterministic test outcome. Listing 1 represents the flaky test case code and Listing 2 represents the test execution logs. The variable **`WriteTData&Verify`** (line 2 in Listing 1) set to the value 1337 (the reason for this specific value cannot be stated due to non-disclosure agreement). After a short delay, the reset command is sent to the hardware and data (i.e., 0) is written to a variable named **`testValue`** (i.e., not showing in Listing 1—some variables and logic have been removed due to non-disclosure agreement) on the device to compare with the value of **`WriteTData&Verify`**. This **`testValue`** is then accessed to ensure that value and device have been reset in the **`WaitForTData`** (line 6 in Listing 1). Listing 2 Line 3 and 4 represent the expected test value and actual test value which are different, resulting in test case failure.

TABLE 4 Information about test smells inducing test flakiness in cases A and B

Case	Received tests	Confirmed flaky tests	Test cases exhibited test smells	Test smells	Frequency
A	1609	0	402	Async wait	369
				Randomness	24
				Input/output	9
B	150	30	30	Async wait	15
				Platform dependency	8
				Time	7

In the context of case A, we identified 369 (91%) test cases out of 402 that contain test smells related to async wait category. These test cases use instructions such as **`Sleep`** (91 occurrences), **`Wait`** (36 occurrences), **`Timeout`** (29 occurrences), and **`Network`** (213 occurrences) to wait for definite period of time for an external resource to be available for use. In this case, the resource takes more time to become available, the test cases fail. Another example of test flakiness with **`Network`** is when test cases interact with Internet. Tests whose execution depends on Internet can be flaky because the network is a resource that is hard to control [7].

Listing 1: Hardware Reset Flaky Test Code

```
RESET PROCEDURE
WriteTData&Verify     (1337)
Delay for    500ms
GENI     Run the RESET command
Delay for    500ms
WaitForTData     (0)
```

Listing 2: Execution Log of Hardware Reset Flaky Test

```
Log      #11      {Log} Waited for 00:00:00 to recieve the test data
Log      #11      {Log} Waited for 00:01:00.4930 to recieve the test data
Log      #13      {Log} Expected Test Value = 0
Log      #14      {Log} Actual Test Value = 1337
Fail     #15      {Log} Name: Set Verdict
Source: Fail
Result: Timeout while waiting for test data
Timeout while waiting for test data
```

4.2.2 | Platform dependency

Platform dependency was observed only in case B. Testbeds are used to study system modules and interactions in order to gain detailed insight into the essence of the real system [34]. Testbeds are built of prototypes and pieces of real system components [34]. Company B described the reason of flakiness as multiple tests were observing lower motor speed than expected which was caused by air bubbles in the water. The water pressure in the testbed had fallen to almost 0 bar. The testbed and tests were designed to be at around 1.5 to 2.5 bar. The pressure slowly decreased over time until it reached to 0. It took around 2 months to fall from 2 bar to 0 bar. To resolve this issue a testbed self-test, as presented in Listing 3, is now run once a day as part of the test plan. This self-test checks the integrity of the testbed and sets or resets the pressure if it gets too low.

Listing 3: Flaky Test Code for Testbed Instability

```
if (CurrentPressure_bed < TestMinPressure)
MeanPressure = (TestbedMaxPfressure + TestbedMinPressure) / 2
Log ‘Pressure too low:’ + CurrentPressure_bed + ‘ bar’
Log ‘Attempting to set pressure to:’ + MeanPressure + ‘ bar’
IO_SetSystemPressure     (MeanPressure)
Delay for    30000
IO_SetSystemPressure     (TestbedMinPressure)
```

4.2.3 | Time

This type of flakiness was identified in case B. The test to check the real time clock on the device failed on several occasions. A Factory **`reset`** in the test caused the time counter to advance by 1 h. Since the test also sets Daylight Savings to be enabled, the test case fails twice a year. In addition to this, the delay to re-save time on the clock varies by 0.3 s affecting the time comparisons between expected and actual values. Listing 4 represents the log for the flaky test that involves clock manipulation in the embedded systems.

Listing 4: Execution Log of Hardware Reset Flaky Test

```
Fail #13 {Log} Name: Compare Expected Run Time Differences
Source: ScriptRunTime
Expected: to be 00:03:01.0400000 (+/−) 00:00:00.3000000
Value: 00:03:00.5822079
Result: Value too low. Deviation: −00:00:00.4577921
```

4.2.4 | Randomness

In case A, 24 (5%) test cases contain test smells that are related to randomness. Test cases are flaky because it use a random number generator in a way that did not function for the test [33]. This type of flakiness occur due to a use of randomized seed when testing the correctness of batch normalization. The mitigation strategy is to switch to a user defined seed. Test cases in case A did not use **`Seed`** function when using a random generator.

4.2.5 | Input/output

9 (2%) test cases contain test smells that are related to Input/Output (IO) in case A. This type of flakiness is related to file descriptors or resources that do not exist [33]. We observed 7 cases where tests try to access external resource without checking if the resource exists or not. Another 2 cases when tests open a file and finish execution without closing it. Another test that would try to open the same file would pass or fail depending on whether the file was already garbage collected or not.

4.3 | Perceived factors and their effect—RQ2

This section explains each perceived factor in detail together with the practitioners' quotes and assigned category. A factor according to an online dictionary [39] means: *"a fact or situation that influences the result of something."* In the context of our study, the "a fact or situation," in the aforementioned definition, represent the identified perceived factors, the "influences" represent reduction or increase, and the "event" represents test flakiness. During data collection, we did not intend to capture dependencies between perceived factors. For example, the factors **`Perseverance to reduce test flakiness`** and **`Team experience in handling test flakiness`** will eventually lead to simple and robust test cases, but participants did not reveal their experiences/perceptions about these types of dependencies. This is why all factors, in this study, were reported independently of each other.

Table 5 presents the category, the identified perceived factors, and their mapping to the available literature. For example, "simple test case" (i.e., one of the identified perceived factors in this study) is mentioned as "simplicity" in [35] in the context of identifying good tests. As presented in Table 5, (1) six perceived factors (i.e., test code) are reported in the literature as qualities of good test cases, (2) three perceived factors are related to a system under test, (3) eight perceived factors concern CI/Test infrastructure, and (4) two perceived factors were categorized as organization-related.

4.3.1 | Test code-related factors

All of the factors in this section are categorized as "test code" because these factors have been mentioned in the literature in the context of test case quality. These factors are perceived by practitioners to affect test flakiness. In addition to practitioners' quotes, we present the definition of each perceived factor together with its perceived effect (i.e., reduce or increase) on test flakiness.

Test case simplicity:$^{\rightarrow (-)\ \textit{test flakiness}}$
As said by Bowes et al. *"Keep it simple, keep it safe"* [35], test case simplicity refers to the number of assertions per test case [35]. Participants in companies A and B perceive test case simplicity as a factor that reduces test flakiness. They write simple test cases, for example, 1–2 assertions per test case. These companies claimed to have very low or no test flakiness, as presented in Table 6. One of the participants said: Testers, in these particular companies, follow the

TABLE 5 Identified factors and their mapping to available literature and category

Category	Identified perceived factors	Mapping of identified perceived factor to available literature
Test code	Test case simplicity	Test case simplicity [35]
	Test case size	Single responsibility [35]
	Test case age	Obsolete test case [21,35]
	Test case robustness	Test case robustness [36]
	Test case independence	Test (in)dependency [35]
	Test case smelliness	Test case smelliness [36]
System under test	System under test/Test case execution time	Test case and test suite timeout [21]
	Requirements clarity	Asses conformance to regulation[37,38]
	Avoiding testing of a complex feature	-
CI/Test infrastructure	Automated test case inspection	-
	Testing for test flakiness at different stages	-
	CI instability	CI instability [21]
	Undermining network infrastructure	Robust network [36]
	Advanced test result reporting	Advanced test results reporting [21,36]
	Rerunning test case	Rerunning test case [27]
	Handler outside test cases	Too much setup code [21]
	Environment understanding	Lack of insight into the systems [21]
Organization-related factors	Perseverance to reduce test flakiness	Assure quality [37]
	Team experience in handling test flakiness	Experienced team [37]

test design principle of 1–2 assertions per test case. They reported that when a test case tests 1–2 things, its perceived output is more deterministic as said by one of the participants:

> "What is the definition of simple is of-course, not obvious but mostly, you can say it has one assertion and on average less than two assertions to have a deterministic output"

Companies C–E claimed to write complex test cases with more than 4–6 assertions per test case and reported high test flakiness.

Test Case Size:$^{\rightarrow\ (+)\ \textit{test flakiness}}$

Test case size refers to the number of uncommented lines in the test code [35]. Company A and B have a design principle of writing small test cases e.g., test cases with few lines of code. A small test case is not necessarily a simple test case (see test case simplicity above) and this is why we label it as a unique perceived factor. One of the participants commented about small test case:

> "The small test will know how to test the specific feature in a specific way and all tests should follow this [size] rule so that the output will always be deterministic"

Test case size was also listed as a perceived issue by another participant in Company B:

> "When a test case tests so many features, it grows complex and becomes flaky [due to complex code and hardcoded conditionals] and no one knows where things are wrong because people, in general, do not want to spend time on complex test cases. These test cases are pending on tester's to-do list because we do not delete test cases if they are flaky. Thus we have more flaky tests every time we run the build"

One of the participants from company A shared previous experience:

> "It is important because earlier all tests were written by testers and developers have all types of long and crazy code and that is why many tests were non-deterministic in our settings"

One of the participants from company B shared the strategy of writing small test cases to reduce test flakiness:

> "Earlier, we had crazy code in the test suite, written by very old [who have left the company] testers and we had lots of non-determinism in test output. Then our team started dividing test cases into smaller test cases and we are happy because we have less flaky tests now"

Companies C–E reported that they do not have small test cases (e.g., Table 6) and this could be one the reasons, as reported by mentioned companies, that they have high test flakiness.

Test case age:$^{\rightarrow (+)\ \textit{test flakiness}}$

Older test cases refer to those test cases where changes in one part of the test case were not propagated to other parts of test cases [21]. For example, the size of a test case grew over time without adjusting the maximum execution time leading to non-deterministic output. There could be several reasons for non-determinism such as unable to download prerequisites, a test not producing output for a fixed amount of time and later, killed by the execution system assuming that the test stalled [21]. One of the participants from company A stated:

> "[…] When it seems to be product issues, for example, the product's functionality has been changed long ago and we have not updated the test case […]. The test case starts changing its outcomes on different occasion leading to increasing flakiness"

The "*Changes in the product*," mentioned by the developer in the above quote, do not refer to recent changes followed by regression testing but the functionality of the specific product becoming more complex, over time, while the test case still tests the smaller part of the original functionality. Upon different/unexpected behaviour of the specific product, not known to a new tester, the test case may fail intermittently. As presented in Table 6, participants from three companies perceived test case age as a factor that affects test flakiness.

TABLE 6 A comparison of perceived factors within investigated companies and Eck et al.

The perceived impact of flakiness		Reported flakiness					
		None	Very Low	High	High	High	High
		Case Companies					
Category	Identified Perceived Factors	A	B	C	D	E	Mozilla
Test Code	Test Case Simplicity	✓	✓	×	×	×	-
	Test Case Size	✓	✓	×	×	×	-
	Test Case Age	✓	✓	×	×	✓	✓
	Test Case Robustness	✓	✓	×	×	×	-
	Test Case Independence	✓	✓	×	×	×	✓
	Test Case Smelliness	✓	✓	✓	✓	✓	✓
System Under Test	System Under Test/Test Case Execution Time	✓	✓	×	×	✓	✓
	Requirements Clarity	✓	×	×	×	✓	✓
	Avoiding Testing of a Complex Feature	✓	×	-	-	-	-
CI/Test Infrastructure	Automated Test Case Inspection	✓	×	-	-	-	-
	Test for Flaky Tests at Different Stages	✓	✓	-	-	-	-
	CI Instability	✓	✓	-	-	-	✓
	Undermining Network Infrastructure	✓	×	×	×	×	-
	Advanced Test Result Reporting	✓	✓	×	×	×	✓
	Rerunning Test Cases	✓	✓	✓	✓	✓	✓
	Handler Outside Test case	✓	×	×	×	×	✓
	Environment Understanding	✓	✓	✓	✓	✓	✓
Organization-Related	Perseverance to Reduce Test Flakiness	✓	✓	×	×	×	-
	Team Experience in Handling Test Flakiness	✓	✓	×	×	×	✓

Note: The symbol "✓" represents that the perceived factor was mentioned by the company and frequently observed. The symbol "×" represents that the perceived factor was mentioned by the company but not practiced at all. The symbol "-" represents that no information is provided about perceived factor.

Test case robustness:$^{\rightarrow\ (-)\ \textit{test flakiness}}$
Robustness, as provided by IEEE [40], is the ability to cope with errors during execution and with erroneous input. Company A and B reported that robust test cases should be written by assuming that flaky behaviour exists in/around the system under test. Besides, robust test cases should address issues regarding integration with external libraries or configurations (e.g., network or port issues), thereby leading to reduced test flakiness. One of the participants from company B stated that

> "Whenever we see flakiness due to network problems, we did not disable test cases. We did not do anything. That is why we saw a lot of flaky tests. But later we started building robust test cases to handle a bad network and unexpected situations and then we had less test flakiness"

One of the participants from company A said:

> "In our team, we design and review test cases with detailed knowledge of the product, its dependencies and in which environment the test case will run. We know what goes in and what goes out and how much the test has to test"

A robust test case can only be developed if testers have clear requirements of a product's functionality and detailed test design and review process, as discussed above and under *"requirements clarity."* There is a threat that a robust test case can become complex if the goal is to address all uncertainties. As one participant shared:

> "I will say that it should not be the test case [that contains complexity] but something else- [i.e. handler outside test case]"

Test case independence:$^{\rightarrow\ (-)\ \textit{test flakiness}}$
This factor complements what other researchers [7,15,41] have concluded: higher dependencies among test cases increase test flakiness. Company A and B claim to have a design principle of having no dependency among test cases and they reported that this is one of the reasons that we have none to very low test flakiness as presented in Table 6. One of the participants from company A commented:

> "You can take away some of the test cases so you have very little test case dependency. The test case is not allowed to depend on another test and that is how we have reduced test flakiness"

The test cases in companies (C–E) as presented in Table 6 have higher dependencies among test cases. The reported reasons were (1) Test suites are not maintained properly, (2) Lack of design principles in writing independent tests, or (3) Tests being written by any member of the software team.

Test smelliness:$^{\rightarrow\ (+)\ \textit{test flakiness}}$
Test smells are poorly designed tests and negatively affect the quality of test suites [42]. Test smells are most important causes of test case flakiness reported in literature as well as said by one the participants:

> "I would say that test code, on average, is as crappy as source code but people pay attention to the source code but for not to test code and they add too many test smells in test code leading to increased test flakiness"

All investigated companies shared a concern that test smells can often be sneaked into test cases when new employees—who are not aware of test design principles—write test code. All the investigated companies with none to high flakiness have observed test flakiness due to the test smells as presented in Table 6. The test smells found in the test cases in investigated companies are presented in Section 4.2.

4.3.2 | System under test-related factors

The factors in this section are mentioned in prior work as production code or system under test.

System under test/test case execution time:$^{\rightarrow\ (-)\ \textit{test flakiness}}$
This factor relates to software under test/test suite timeout, identified by Eck et al. [21], where a test suite has grown in size (e.g., lines of code and complexity), while test execution time limit is still what was set when the test case was

written thus leading to non-deterministic output. Some functionalities of a product, in companies A, B, and E require longer execution times for the system under test. Test cases that were written for these situations became flaky due to several reasons such as memory shortage or heap size reaching the limit or test case timeout. One of the participants explained:

> "During longer executions of system under test, even though the software was not changed, the longer running test case becomes flaky. You could call it a kind of long term testing or long term reliability testing and for this reason, flaky behavior was detected, which was not possible without it [longer execution]"

Requirements clarity:$^{\rightarrow (-)\ \textit{test flakiness}}$

Lack of clarity about the system requirements increases the ambiguity of what exactly the system does, thus leading to unclear tests design requirements [43]. Participants believed that test flakiness can be reduced if testers put enough effort during test design (i.e., valid input/output values within the assertion range), with clear requirements and what is expected from the test. "Lack of clear requirements" during test design process along with the undefined project are mentioned as the perceived causes of the increase in test flakiness in companies B–D. One of the participants from company B said that:

> "For the project we have now, where we lack requirements, the reason for test flakiness was unclear requirements. This [test flakiness] is for an undefined project and unclear requirements"

Companies A and E stated that they assign skilful and experienced people during requirements gathering and the test design process whereas companies B–D reported that they form ad-hoc teams for the task at hand and welcome anyone from any teams to write test cases.

Avoiding testing of a complex feature:$^{\rightarrow (-)\ \textit{test flakiness}}$

Participants in company A strongly believed that testing of a specific features/requirements lead to test flakiness and if they could avoid testing this specific feature/requirement, test flakiness would be substantially reduced. This belief has been turned into a common practice in company A. As one participant mentioned:

> "We are ignoring feature X and feature Y and then we avoid complex feature testing and we do not have test flakiness"

One of the participants shared that

> "When we test the functionality of 'restart device', we do not get acknowledgment every time that the device has been restarted. In this case, sometimes tests pass and sometimes tests fail, we stopped testing this feature on all devices […], thus the tests became more deterministic"

One could argue about the trade-offs between risking less test coverage and reducing test flakiness but this is not within the scope of this paper.

4.3.3 | CI/test infrastructure-related factors

The factors in this section are mentioned in prior work as CI or test infrastructure.

Automated test case inspection:$^{\rightarrow (-)\ \textit{test flakiness}}$

Company A reported that if the test case exhibits flakiness, you should write another automated test case that explicitly catches the flakiness in the test. One of the participants commented that

> "Consider that a test A is a flaky test. Then you can write a test B that explicitly catches this flakiness: a test that repeats test A 100 times and explicitly states that it needs to succeed 100 times"

This practice is widely adopted only in company A. Other companies only rerun the tests if they suspect test flakiness. Automated test case inspection is a perceived factor or recommendation to increase the likelihood of the detection of test flakiness. The practice of writing automated test cases to catch flaky tests increase the trust in the final product as mentioned by of the participants:

> "Writing another automated test case that explicitly catches this flakiness require more efforts and time but on the other hand, gives us indications to trust the test suite results and the final product product"

Testing for flaky tests at different stages:$^{\rightarrow (-)\ \textit{test flakiness}}$

Different testing activities (e.g., system or integration testing) should incorporate methods to reveal flakiness in the CI pipeline. The chances to detect test case flakiness increase when you hunt for it at different testing stages. As one of the participants said,

> "The automated regression tests are not the only line of defense against flaky test but we have several instances where we expect the flaky test to be caught on the way"

All investigated companies had observed test flakiness during acceptance, system or integration testing. The study participants had not observed flakiness during unit testing. Company A and B recommended that each testing activity in CI should include mechanisms to detect test flakiness. Company C, D, and E did not pay attention to flaky tests until the system testing.

As claimed by participants in case A, test flakiness detection on multiple stages increases the trust in the test cases:

> "We have many stages of checking if the test case is flaky or not. This is why, we do not have flaky tests in our pipeline and we trust our test suites"

CI instability:$^{\rightarrow (+)\ \textit{test flakiness}}$

Continuous integration platforms are claimed to play a major role in increasing test flakiness [21]. As one of the participants commented,

> "On the Jenkins side, the biggest problem they had is flakiness in Jenkins. When a new release of Jenkins is deployed, and this is invisible to the other users because the users do not care what version of Jenkins they are using and they are running the test and suddenly tests change outcome from pass to fail and they do not know what happened [….]"

This factor has been experienced by all investigated companies and they claimed to invest so much debugging time just to find out that the reason for the change in tests outcome is Jenkins instability. The solution to this type of flakiness is to wait for a stable release resulting in frustrations and delays of the software release.

Undermining network infrastructure:$^{\rightarrow (-)\ \textit{test flakiness}}$

Undermining network infrastructure refers to practices where companies intentionally create worst-case scenarios to test the robustness of the test code and system under test. Company A has implemented the specific practice of undermining network infrastructure. They reported that this is one the reasons that they have no test flakiness. Participants at company A ensure that worst-case scenarios should be assumed when writing test cases for networks, ports, I/O or third-party libraries for deterministic output. As one of the participants from company A mentioned:

> "We were [not now] trying to avoid network issues by making sure that network is always up and running. If you assume that the network (Ethernet/ IP network) is perfect, you will have non-deterministic tests"

Another participant said:

> "We are actively undermining the network, by making it worse, so you design test cases to reveal the real failure instead of trying to clean the ways for tests"

Participants at company A reported that they had inexperienced testers who write Thread.sleep(60ms) to wait if resources are available but if we undermine network infrastructure, more test cases will be revealed that are prone to change their outcomes due to async wait issues.

Advanced test results reporting:$^{\rightarrow (-)\ \textit{test flakiness}}$

A better mechanism to report, log and display test case results helps in detecting test case flakiness as shared by Company A and B. One of the participants mentioned,

> "As soon as Team City fails the test, we have a monitor that says which project is failing and you see which tests have changed their outcome [through special plugins to compare builds' output]. Through that plugin we discover that a test has changed its outcome, so I take a look whether it is due to the device under test or the test case code, then I assign this issue to someone in the team responsible"

Companies A and B regularly use different plugins to generate test result reports that help them identifying flaky tests. Addressing the causes of identified flaky tests eventually reduces flakiness and this was one of the reported reasons by companies A and B that they have none to very low test flakiness. On the other hand, as can be seen in Table 6, companies C–E only depend on Jenkins log files. Participants in companies C–E manually parse the log files to investigate if test cases have changed their outcome between the current and the last build. In addition to identifying the flaky tests, the lack of detailed logs can be challenging to understand the behaviour of test cases and the causes of failures.

Rerunning test cases:$^{\rightarrow (-)\ \textit{test flakiness}}$

Rerunning test cases, as mentioned in [27], is the most widely used technique to detect test flakiness. All the participants stated that re-running test cases over a night or weekend helps detect flakiness:

> "When we write a test case we test it regularly to find out if they are flaky or not [..] After we have written it and integrated it so we run it on all products (approx. 50) each night for 7 days two times. It is $2 \times 7 \times 50 = 700$"

Companies B, C, D, and E use rerunning as the only mechanism to detect test flakiness as compared to company A which deploys another method *'automated test case inspection'* discussed in Section 4.3.3.

Environment handler outside test cases:$^{\rightarrow (-)\ \textit{test flakiness}}$

Company A has a practice of removing complex code from test cases and placing it in helper functions or library routines to reduce test flakiness, as mentioned by one of the participants in company A:

> "We noticed a significant difference in test flakiness when we removed complex code from the test case and placed it into a separate library named [Do-X]. All products can use [Do-X] differently so we have different libraries that provide different [Do-X] functions. The test code only says [Do-X] and depending on the product, we can call the corresponding function from the library"

Participants in company A shared that the other teams can use these external libraries and reduce the risk of writing unreliable (since these libraries are reviewed and maintained properly) and redundant code which can affect the test flakiness. Eck et al. in [21] label this factor as one of the challenges in test flakiness by naming it *"too much setup code."* The individual test case does not need to know if the network or product is not available and should always fail rather than exhibiting non-deterministic behaviour. These external libraries will provide proper test log results such as "test failed" due to no network availability for product X. Company A was the only company that writes external setup code. All other companies write all setup code or handlers in the test case and reporting high test flakiness.

Environment understanding:$^{\rightarrow (-)\ \textit{test flakiness}}$

The word "environment" refers to the system under test, configurations, and integration with third-party libraries. Participants in company A shared that understanding a complex system in detail will lead to a deterministic output of the test cases. However, system can be so complex, as mentioned by one of the participants of company A:

> "It is rather that they [testers] are not seeing things that are changing in the environment and sometimes the source of this change is a timing issue or network and you write test cases that are unaware of these changes thus failing and passing on different occasions leading to non-deterministic behavior"

All the investigated companies reported that they put an extra effort to understand how the system under test and continuous integration machinery works together to reduce test flakiness. Eck et al. [21] mentioned this factor as a *"Lack of insight into the system,"* which can increase test flakiness.

4.3.4 | Organizational-related factors

We observed that the following factors concerns human or organizational perspectives.

Perseverance to reduce test flakiness: $^{\rightarrow\ (-)\ \textit{test flakiness}}$

Perseverance, in this study, refers to practitioners' willingness to find root causes of test flakiness and an ability to not give-up, once the test case was marked as flaky. However, perseverance to reduce test flakiness should also reveal efforts in writing robust/simple test cases, but none of the participants reported it. Company A and B claim to have teams which have perseverance to reduce test flakiness. One of the participants from company B commented:

> "It was a very persistent analysis when people said that we want to know why this test case is changing its outcomes, without any changes in the code base, and we spent days to know the reason for flakiness"

Another participant shared the same experience of being persistent:

> "When we detect flakiness, one or two times we updated the test but the test case was still flaky. Then we also debugged what could be the problem and at the end, we figured out [the reason for test flakiness]. It was hard to find"

Software teams in companies C–E shared the confusion about identifying the person who can take the responsibility to fix the flaky test. Practitioners shared that these confusions happened due to a lack of clear roles and responsibilities concerning test flakiness, as mentioned by one of the participants:

> "Yes, it is usually back and forth between us and developers as they say that this flakiness is due to you [testers] and we say that this flakiness is due to you [developers] and it must be fixed by you"

Team experience in handling test flakiness: $^{\rightarrow\ (-)\ \textit{test flakiness}}$

Previous experiences in test flakiness help as mentioned by one of the participants from company B:

> "Generally I look at test logs for how many tests fail and then I can generally work out what causing it. I can tell easily, due to past experiences [if it is flaky and why]"

Another participant from company B stated:

> "Inexperienced testers—who write Thread.sleep(60ms) and they do not know how these instructions work—can increase test flakiness"

Another participant from company A shared how experience helps in dealing with test flakiness

> "[.….] Depending on how experienced the team is in writing the tests I say most of them think about asynchronous calls and timing issues"

4.3.5 | Tentative mapping of relationships based on perceptions

During data collection, participants shared the perceived effect of the perceived factor such as that some factors can either increase or decrease test flakiness. We grouped these perceived factors based on their effect as represented by Figure 3. Twelve perceived factors out of nineteen are claimed to decrease test flakiness. Three perceived factors are claimed to increase test flakiness whereas four factors affect the ability to detect test flakiness.

4.4 | Practitioner's perceptions versus test artefacts—RQ3

We received a limited data set from the cases (i.e., A and B). Case A provided 1609 test cases which they claimed to be non-flaky. Case A did not provide any test case that was flaky. Case B provided 150 test cases in which 30 test cases were marked as flaky. Given the test artefacts availability, we looked for the evidence of whether or not the developers'

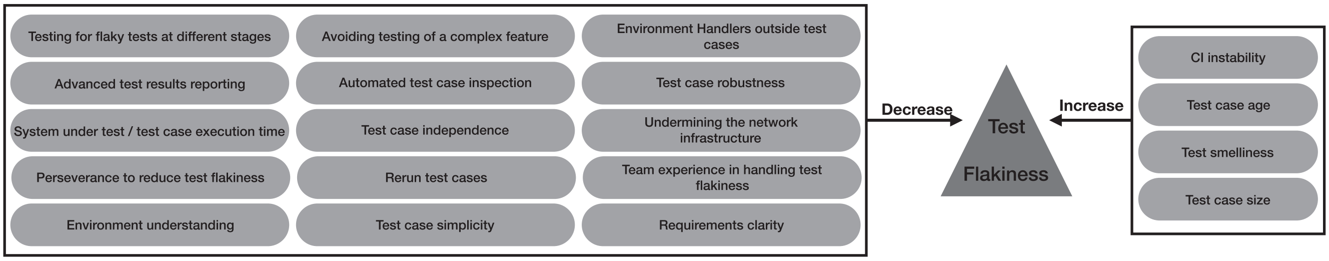
Testing for flaky tests at different stages
Avoiding testing of a complex feature
Environment Handlers outside test cases
Advanced test results reporting
Automated test case inspection
Test case robustness
System under test / test case execution time
Test case independence
Undermining the network infrastructure
Perseverance to reduce test flakiness
Rerun test cases
Team experience in handling test flakiness
Environment understanding
Test case simplicity
Requirements clarity
Decrease
Test Flakiness
Increase
CI instability
Test case age
Test smelliness
Test case size

FIGURE 3 Tentative mapping of relationships based on perceptions

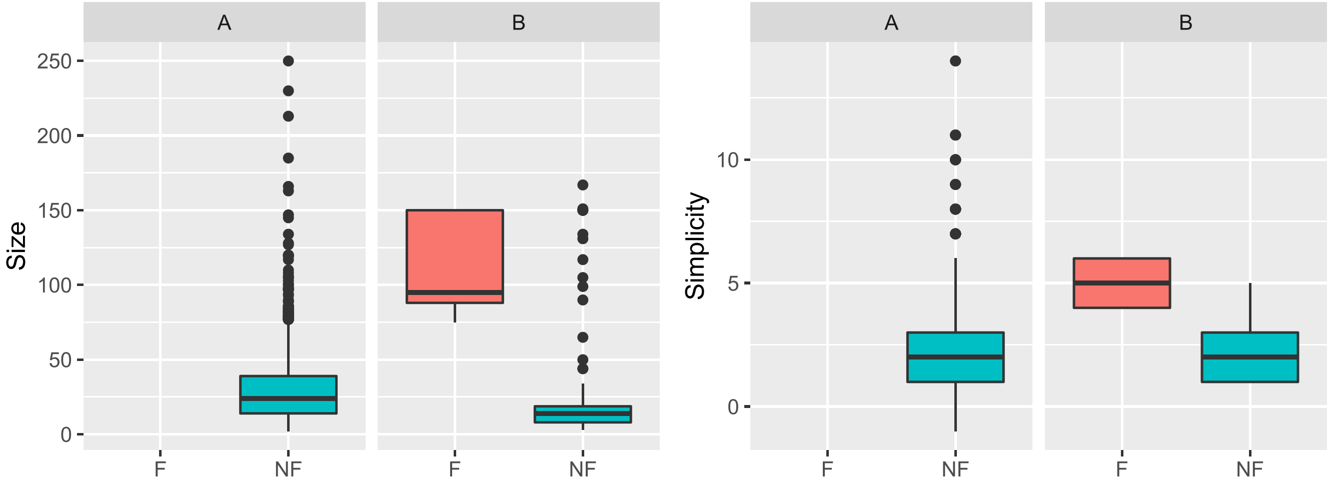

FIGURE 4 Size (i.e., lines of non-comment codes) and simplicity (i.e., number of assertions) in the test artefacts within companies A and B with respect to flaky and non-flaky tests. *Note*: case A did not provide any marked flaky tests and claimed that all tests are non-flaky

perceptions match with what they have marked as flaky or not-flaky. Two perceived factors (i.e., test case size and simplicity) were potential candidates that can be statistically measured through automated scripts. Test case size represents the non-comment lines of code whereas test case simplicity represents the number of assertions in the test case. Figure 4 presents the box-plot for test case size and simplicity within two companies with respect to flaky or non-flaky tests.

We noticed a clear relationship between the test flakiness and test case size in company B. Non-flaky tests in company B consist of less number of lines as compared with the flaky tests, as shown in Figure 4. The median for non-flaky tests in cases A and B is 24 and 35, respectively. The median in flaky tests in case B is 95, which is far more higher than from non-flaky tests. For company A, we did not have flaky tests, but we can see, in Figure 4, that the lines of codes in non-flaky tests in company A is similar to the lines of codes in non-flaky tests in company B. We speculate that this could be one of the reasons that company A has experienced no flakiness.

Similar clarity between test flakiness and test case simplicity was observed in case B. Non-flaky tests in case B consist of less number of assertions as compared with the flaky tests, as shown in Figure 4. The median in the non-flaky tests within cases A and B was 2 whereas the median in flaky tests in case B was 5. Case A has 1–2 assertions in each non-flaky test case except in some cases where the number of assertions reaches 14 (i.e., outliers in Figure 4). Again, we consider this to be the reason for no flakiness in case A.

5 | EVALUATION

We conducted an evaluation with different participants from the companies to understand how important the perceived factors were to practitioners before evaluating each factor's effect. During the first evaluation, the participants ranked the importance of identified factors. Each factor was ranked with respect to their importance with a Likert scale of *Strongly disagree* to *Strongly agree*. Each factor was discussed with all participants to avoid confusion. As presented by Figure 5, 14 out of 19 perceived factors were ranked as *Agree* or *Strongly agree* in terms of importance by all of the

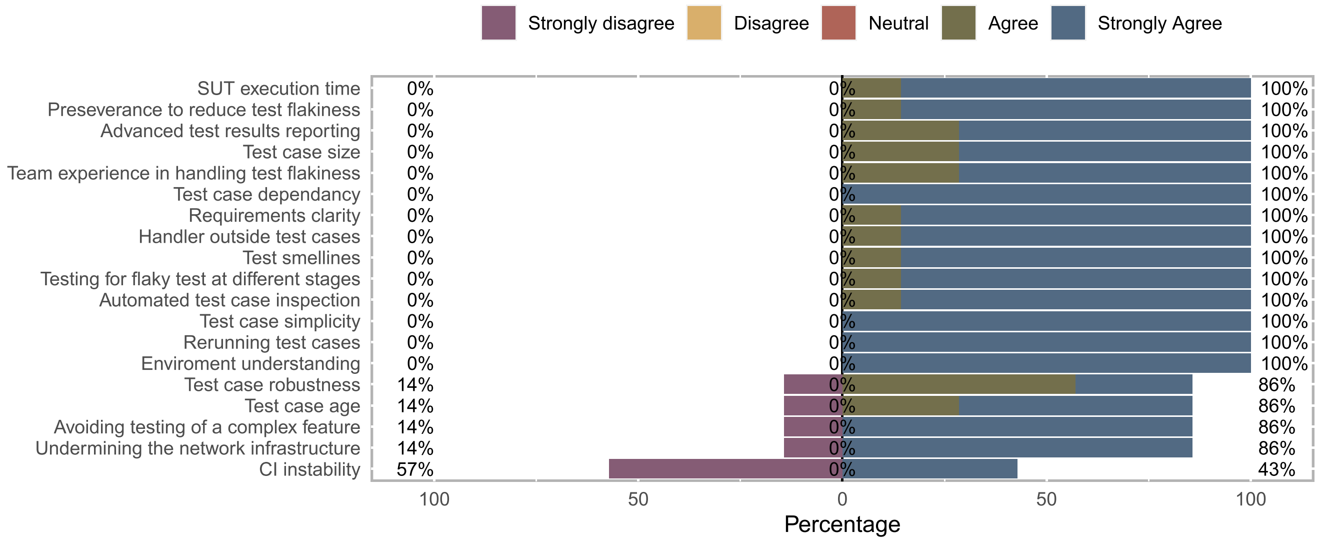

FIGURE 5 Agreement level (in percentage) about importance of factors by participants

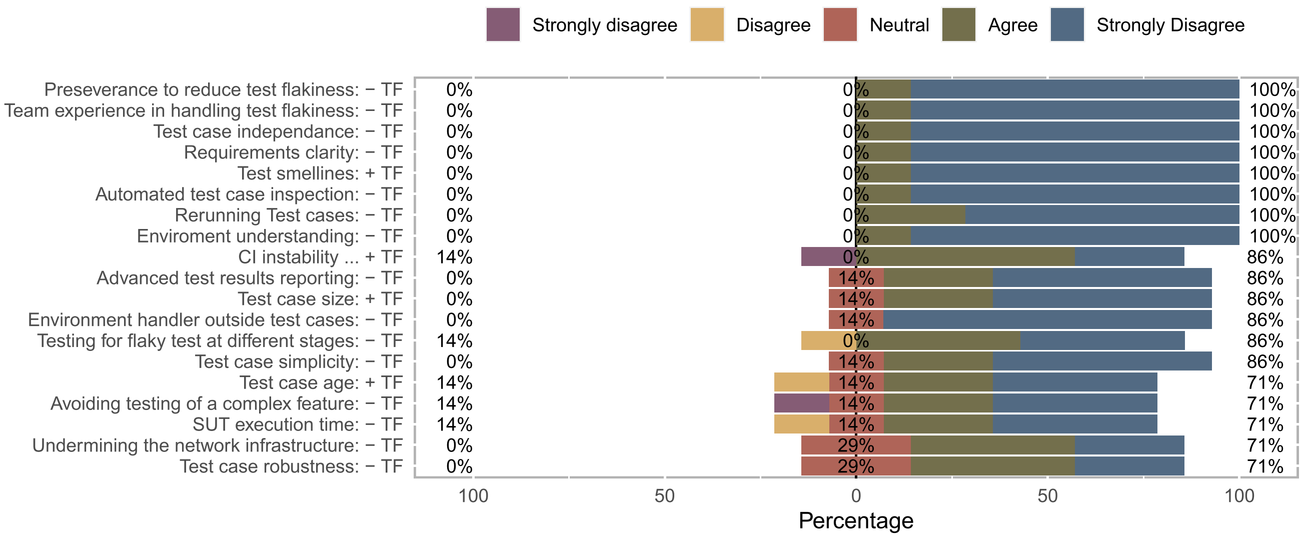

FIGURE 6 Ranking of the effect of identified factors on test flakiness by participants. To improve readability, we used TF to represent test flakiness

participants. Four perceived factors such as “Avoiding testing of a complex feature,” “Undermining network infrastructure,” “Test case age,” and “Test case robustness” were marked as *Strongly disagree* by only one participant whereas only “CI instability” was marked as *Strongly agree* by many participants. We expected this ranking, because what developers perceive as test flakiness in their own organization may not be applicable to other developers in other workspace.

In addition to the evaluation of the perceived factors, we conducted an evaluation for participants' agreement level on the estimated effects of the identified factors on test flakiness based in Likert scale. Figure 6 represents total agreement scores, assigned to each factor's effect by the practitioners. Participants were mostly *Strongly Agreed* or *Agreed* with most of the effects of the perceived factors. We observed only two cases of *Strongly Disagree* and three cases of *Disagree*. The effect of a perceived factor (i.e., CI instability → + test flakiness and avoiding of a complex feature → − test flakiness) was ranked *Strongly Disagree* by only one participant (it is important to note that these participants are different than what were requested for data collection). The effect of a perceived factor (i.e., Testing for flaky tests at different stages → − test flakiness, test case age → + test flakiness, and SUT execution time → − test flakiness) was ranked *Disagree* by only one participant).

6 | DISCUSSION AND IMPLICATIONS

6.1 | Proper roles and responsibilities: avoid blame-game

In addition to what test flakiness is and where it originates from, we observed, during a workshop with two companies, that software professionals at some of the companies are reluctant to take responsibility if test flakiness is detected. Due to the lack of understanding of the actual root causes of test flakiness, practitioners play a "blame-others" strategy as discussed in Section 4.3.4.1. We were informed, during this study, that upon detection of test flakiness, the test is skipped, ignored, or deleted from the test suite by the teams in three companies (C–E). The testing teams in other two companies actually hunt for the root causes of test flakiness. This hunt can be manual (i.e., looking into execution logs, etc.) or automatic (i.e., automated test inspection—Section 4.3.3.1). We suggest that proper roles and responsibilities should be defined within the teams to detect and fix test flakiness as per shared in Section 4.3.4. We found another evidence for lack of proper roles and responsibilities: all the investigated companies, in this study, have a separate department for quality assurance. However, developers were welcomed to write test cases. In some of the cases, the test cases were suggested by external stakeholders (e.g., external companies that develop software for the hardware, developed by the investigated companies).

6.2 | Trust on final product and testing process

We believe that test flakiness (detection, removal, prediction, and assessment) requires immediate attention because non-deterministic tests raise a question: Can I trust my test results, test suite or even the quality of the final product? Flaky tests decrease the confidence in the final product as well as leading to unstable continuous integration or delivery pipeline. Deleting or skipping flaky tests increase the risk that some defects in the software can be released without being detected by a test or a tester. However, this risk needs to be traded against the goal of having fast feed-back, which supports productivity. With proper test-case selection techniques the risk can be mitigated [44]. During the study, we observed that all companies dedicate enough resources for the testing activities but only two companies dedicate proper resources for detecting and fixing flaky tests. We speculate that the increase in flaky test is not due to limited resources but to lack of deep understanding of real root causes of test flakiness.

6.3 | Test flakiness investigation in different contexts

We observed that the flakiness of system tests in embedded system software might be different from stand-alone software that does not require specific hardware. The existence of flakiness in web applications might be different as compared to the desktop systems. Unfortunately, At the time of writing this article, research about test flakiness in embedded systems is scarce. This is why, we attract researchers and practitioners attention towards this necessity We concluded that test flakiness has a relationship with the contexts (i.e., open-source, closed-source, web programming, embedded system, etc.), in which it has been investigated.

6.4 | Test flakiness prevention by test design

Our quantitative analysis shows that 80% of the investigated companies have test case review processes, but none of them have any guidelines to prevent flakiness during the test design process. We suggest that companies should assign special efforts and guidelines to prevent test flakiness at early stages of software development life cycles. All of the identified factors have been discussed in the literature in different contexts (e.g., see Table 6) leading to a conclusion that what practitioners perceive as factors affecting test flakiness can be addressed using the published design and review guidelines in their test design practice to accomplish good quality tests.

We observed, during workshops, that little is known about test flakiness (causes and mitigation strategies) among practitioners in companies C–E and they are struggling to understand the root causes of test flakiness. We suggested to all companies that labelling test case as flaky would be a good start to maintain the database of flaky tests within the companies. The organization can learn from flaky test database and these lessons can reflected in the test design process so flaky tests can be prevented. Developers who ignore or delete flaky test do not only increase the risk of untested functionality but lose the opportunity to learn what caused the test flakiness resulting in repeating the same mistake again in future.

Prior research has focused on (1) the open-source industry, and (2) detecting test flakiness after test suite execution. However, there is a strong need to prevent and predict test flakiness.

6.5 | Flaky test perception and reality

We have observed different perceptions from the very beginning when we conducted the online workshop and site visits for data collection. Different participants had different perceptions regarding what flakiness is and whether we should discuss test flakiness, source code flakiness, or environment flakiness. Some professionals in the study consider the problem of test flakiness as a philosophical issue such as one of the participants stated that *It is a responsibility of the test to not to be flaky, even if you have flakiness in the SUT [….] I hope it will not take too much time to discuss test flakiness as this can be philosophical discussion about what is flakiness and what is not. You can define the flakiness as a point of the observer or implementer*. One the other hand, it seems that given the data, people agree on most of the factors. This might be positive, showing that there is a large portion of flakiness that can be agreed upon with fairly little effort that can facilitate company-to-company experience exchange.

We concluded that two of the perceived factors (i.e., test case size and simplicity) were reflected in the test artefacts (see Section 4.4). Thus, practitioner's perceptions play an important role in defining, maintaining, controlling, or reducing test flakiness within the company.

We have also observed differences in practitioners' perceptions of factors when we conducted the workshop to test our findings. The factor "Test case robustness" was discussed for a long time among the participants during the meeting, where some participants did not agree that this factor affects the test case flakiness. Initially, participants categorized it as *Strongly Disagree*. Similar discussion was raised among participants for "CI instability." These factors was initially thought to be specific to one company, but upon discussion, participants from other companies also changed their opinion and marked it as *Strongly Agree*. This change in opinion motivates our claim for conducting more research in capturing the tacit knowledge within companies in the field of test flakiness. Listening to these arguments during the workshop, we learned that participants' perceptions about test case flakiness are likely to influenced by what they have heard and observed, and experienced in the workplace.

6.6 | Test smells are not the only indicator of test flakiness

Although company A reported no test flakiness but their test cases exhibited test smells as shown in Table 4. Almost all of the test smells were related to asynchronous wait. Several studies have reported *asynchronous wait* as a major root cause of test flakiness [7,22] but we observed that the presence of this test smell does not necessarily represent test flakiness, but test flakiness might be caused by the unexpected behaviour of the external environment, thus requiring more investigation about flakiness in production code, infrastructure and external resources.

6.7 | Test smells: open-source versus closed-source industries

During the survey (see Figure 2h), all the participants from five companies ranked *Async wait* and *Configuration and dependency issues* as the frequent root causes to test flakiness. However, test smells such as *Concurrency*, *Floating point operations*, *Resource leak*, *Source code defect* and *Unordered collection* were not or very little experienced by the participants due to the fact that real-time properties of systems has been heavily researched, so there is already knowledge of how to anticipate these issues and fix them.

These test smell distributions are very different from what other studies with open-source software [1,7,21] have reported. We speculate that the dedicated testing teams tend to educate themselves about common root causes and avoid making the same mistakes in the future. As far as the *Async wait* and *Configuration and dependency issues* are concerned, these test smells are mostly associated with external factors such as availability of resources thus limiting the human control over test flakiness.

7 | VALIDITY THREATS

7.1 | Internal validity

An internal validity threat could be that participants did not understand our coding and its representation correctly. We tried to reduce this by conducting a workshop in person at our university, allowing us to explain these factors and their effects to attendees. In addition to this, we dedicated some time for questions related to these factors and their effects.

7.2 | Construct validity

The main purpose of addressing construct validity is to capture as much as possible of the available information to avoid all sorts of bias. We have tried to address construct validity threat by conducting both workshops (i.e., data collection and data validation) with different participants. We tried to reduce the researchers' bias by involving all 3 researchers in the design of the workshop and of the questions.

7.3 | External validity

External validity refers to the extent to which it is possible to generalize the findings, as well as the extent to which the findings are of interest to other practitioners beyond those associated with the specific case being investigated [45]. We tried to eliminate the external validity threat by selecting five different companies that work in different domains.

7.4 | Reliability

Two researchers, at minimum, were involved to review the research protocol and study results. We continuously iterated the survey/workshop questions to reach to a mutual agreement. We described the method used for conducting this study to make it easier for researchers and practitioners to understand the details of this study.

8 | CONCLUSION

The presence of flaky tests in test suites raise concerns over product quality. They affect the confidence in the product as well as frustrate practitioners as tests change outcomes without code updates. The test flakiness problem requires practitioners' and researchers' focus on techniques/tools/methods/guidelines /approaches that prevent test flakiness, rather than detecting it after test suite execution, similar to any disease where we take precautionary measures before it spreads. It is equally important to raise awareness among practitioners about what causes test flakiness and how to address it efficiently (RQ1). After a manual and automated analysis of the test cases code of flaky tests, we reported the root causes of test flakiness experienced by professionals (RQ1). We attempted to capture a practitioners' perceptions of what they think test flakiness is and what factors affect it. We identified 19 perceived factors that either increase or decrease test flakiness (RQ2). The identified perceived factors were categorized as *Test Code*, *System Under Test*, *CI/Test Infrastructure*, and *Organization Related*. Although the perceived factors mentioned in Section 4.3, have been very helpful for the investigated companies to reduce test flakiness, we concluded that what practitioners perceive as factors affecting test flakiness are, in reality, properties of a good test case (i.e., "simple test case," "small test case," and "robust test case" and others have been categorized as "simplicity," "single responsibility," and "robust test case" respectively, as part of a good test case (e.g., see Table 5) and well defined software practices. The problems related to test flakiness require more understanding of current perceptions or reality among practitioners. In addition, two of the identified perceived factors (i.e., test case size and simplicity) were observed in the test artefacts which represent that the practitioner's perceptions are important to consider to reduce test flakiness. We concluded that test flakiness should be investigated in different contexts to extend current knowledge. After survey analysis, we determined that "Asynchronous wait" and "configuration and dependency issues" have been mentioned as major root causes of test flakiness followed by "test order dependency."

ACKNOWLEDGEMENTS

The authors would like to thank the participants in our multiple case study for their availability to help us in data collection and data validation and to ask clarifying questions about the data as well as the engaging and insightful discussions. This work was supported by Linkoping University and project 30 on Aspects of Automated Testing of Software center.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from Software center. Restrictions apply to the availability of these data, which were used under license for this study. Data are available from the author(s) with the permission of Software center.

ORCID

Azeem Ahmad https://orcid.org/0000-0003-3049-1261

REFERENCES

1. Fowler M. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html. Accessed [2019-04-15 18:52:30].
2. King TM, Santiago D, Phillips J, Clarke PJ. Towards a Bayesian network model for predicting flaky automated tests. In 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE Comput. Soc: Lisbon, 2018. p. 100–7.
3. Leong C, Singh A, Papadakis M, Traon YL, Micco J. Assessing transition-based test selection algorithms at Google. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '19. IEEE Press: Montreal, Quebec, Canada, 2019. p. 101–10.
4. Micco J. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed [2019-04-15 18:48:16]
5. Labuschagne A, Inozemtseva L, Holmes R. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. ACM: New York, NY, USA, 2017. p. 821–30. event-place: Paderborn, Germany.
6. Hilton M, Nelson N, Tunnell T, Marinov D, Dig D. Trade-offs in continuous integration: Assurance, security, and flexibility. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. ACM: New York, NY, USA, 2017. p. 197–207. event-place: Paderborn, Germany.
7. Luo Q, Hariri F, Eloussi L, Marinov D. An empirical analysis of flaky tests. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014. ACM: New York, NY, USA, 2014. p. 643–53. event-place: Hong Kong, China.
8. Rainer A, Hall T, Baddoo N. Persuading developers to 'buy into' software process improvement: Local opinion and empirical evidence. In Proceedings of the 2003 International Symposium on Empirical Software Engineering, ISESE '03. IEEE Computer Society: Washington, DC, USA, 2003. p. 326.
9. Laukkanen E, Paasivaara M, Arvonen T. Stakeholder perceptions of the adoption of continuous integration—A case study. In 2015 Agile Conference, 2015. p. 11–20.
10. Sun W, Marakas G, Aguirre-Urreta M. The effectiveness of pair programming: Software professionals' perceptions. IEEE Softw. 2016;33(4): 72–9.
11. Ebert F, Castor F. A study on developers' perceptions about exception handling bugs. In 2013 IEEE International Conference on Software Maintenance, 2013. p. 448–51.
12. Shah H, Gorg C, Harrold MJ. Understanding exception handling: Viewpoints of novices and experts. IEEE Trans Softw Eng. 2010;36(2): 150–61.
13. Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD. Do they really smell bad? A study on developers' perception of bad code smells. In 2014 IEEE International Conference on Software Maintenance and Evolution, 2014. p. 101–10.
14. Camacho CR, Marczak S, Cruzes DS. Agile team members perceptions on non-functional testing: Influencing factors from an empirical study. In 2016 11th International Conference on Availability, Reliability and Security (ARES), 2016. p. 582–9.
15. Percival J, Harrison N. Developer perceptions of process desirability: Test driven development and cleanroom compared. In 2013 46th Hawaii International Conference on System Sciences, 2013. p. 4800–9.
16. Tan H, Tarasov V. Test case quality as perceived in Sweden. In 2018 IEEE/ACM 5th International Workshop on Requirements Engineering and Testing (RET), 2018. p. 9–12.
17. Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. Are test smells really harmful? An empirical study. Empir Softw Eng. 2015;20(4): 1052–94.
18. Wan Z, Xia X, Hassan AE, Lo D, Yin J, Yang X. Perceptions, expectations, and challenges in defect prediction. IEEE Trans Softw Eng. 2018; 46:1241–66.
19. Zou W, Lo D, Chen Z, Xia X, Feng Y, Xu B. How practitioners perceive automated bug report management techniques. IEEE Trans Softw Eng. 2018;46:836–62.
20. Abad ZSH, Ruhe G, Bauer M. Task Interruptions in requirements engineering: Reality versus perceptions!. In 2017 IEEE 25th International Requirements Engineering Conference (RE), 2017. p. 342–51.
21. Eck M, Palomba F, Castelluccio M, Bacchelli A. Understanding flaky tests: The developer's perspective. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019. 2019. p. 830–40. arXiv: 1907.01466.
22. Thorve S, Sreshtha C, Meng N. An empirical study of flaky tests in Android apps. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018. p. 534–8.
23. Morn J, Augusto C, Bertolino A, de la Riva C, Tuya J. Debugging flaky tests on web applications:. In Proceedings of the 15th International Conference on Web Information Systems and Technologies. SCITEPRESS - Science and Technology Publications: Vienna, Austria, 2019. p. 454–61.
24. Morn J, Augusto C, Bertolino A, Riva CDL, Tuya J. FlakyLoc: Flakiness localization for reliable test suites in web applications. J Web Eng. 2020;2:267–96.
25. Dong Z, Tiwari A, Yu XL, Roychoudhury A. Concurrency-related flaky test detection in Android apps, 2020. arXiv:200510762 [cs], arXiv: 2005.10762.
26. Mascheroni MA, Irrazbal E. Identifying key success factors in stopping flaky tests in automated REST service testing. J Comput Sci Technol. 2018;18(02):e16.
27. Bell J, Legunsen O, Hilton M, Eloussi L, Yung T, Marinov D. DeFlaker: Automatically detecting flaky tests. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018. p. 433–44.
28. Gambi A, Bell J, Zeller A. Practical test dependency detection. In 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018. p. 1–1.
29. Dutta S, Shi A, Choudhary R, Zhang Z, Jain A, Misailovic S. Detecting flaky tests in probabilistic and machine learning applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020. Association for Computing Machinery: New York, NY, USA, 2020. p. 211–24. https://doi.org/10.1145/3395363.3397366

30. Lam W, Godefroid P, Nath S, Santhiar A, Thummalapenta S. Root causing flaky tests in a large-scale industrial setting. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019. Association for Computing Machinery: New York, NY, USA, 2019. p. 101–11. https://doi.org/10.1145/3293882.3330570
31. Shi A, Bell J, Marinov D. Mitigating the effects of flaky tests on mutation testing. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019. Association for Computing Machinery: New York, NY, USA, 2019. p. 112–22. https://doi.org/10.1145/3293882.3330568
32. Strauss A, Corbin J. Basics of qualitative research: Techniques and procedures for developing grounded theory, 2nd ed. Sage Publications, Inc: Thousand Oaks, CA, US, 1998.
33. Sbom A. Studying test flakiness in python projects: Original findings for machine learning, 2019. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-264459
34. Fortier PJ, Michel H. Computer systems performance evaluation and prediction. Butterworth-Heinemann: USA, 2002.
35. Bowes D, Hall T, Petric J, Shippey T, Turhan B. How good are my tests? In 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), 2017. p. 9–14.
36. Deursen A, Moonen LMF, Bergh A, Kok G. Refactoring test code, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 2001.
37. Kaner C. What is a good test case? 2003. Software Testing Analysis & Review Conference (STAR) East, Orlando, FL, May 12-16. p. 16.
38. Beer A, Junker M, Femmer H, Felderer M. Initial investigations on the influence of requirement smells on test-case design. In 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), 2017. p. 323–6.
39. Factor definition and meaning | Collins English Dictionary. https://www.collinsdictionary.com/dictionary/english/factor
40. IEEE Standard Glossary of Software Engineering Terminology, 1990. IEEE Std 61012-1990, 1–84.
41. Lam W, Oei R, Shi A, Marinov D, Xie T. iDFlakies: A framework for detecting and partially classifying flaky tests. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019. p. 312–22.
42. Garousi V, Küçük B. Smells in software test code: A survey of knowledge in industry and academia. J Syst Softw. 2018;138:52–81.
43. Dar HS. Reducing ambiguity in requirements elicitation via gamification. In 2020 IEEE 28th International Requirements Engineering Conference (RE), 2020. p. 440–4.
44. de Oliveira Neto FG, Ahmad A, Leifler O, Sandahl K, Enoiu E. Improving continuous integration with similarity-based test case selection. In Proceedings of the 13th International Workshop on Automation of Software Test, AST '18. ACM: New York, NY, USA, 2018. p. 39–45. http://doi.acm.org/10.1145/3194733.3194744
45. Runeson P, Host M, Rainer A, Regnell B. Case study research in software engineering: Guidelines and examples (1st edn.) Wiley Publishing, 2012.

How to cite this article: Ahmad A, Leifler O, Sandahl K. Empirical analysis of practitioners' perceptions of test flakiness factors. Softw Test Verif Reliab. 2021;e1791. https://doi.org/10.1002/stvr.1791