Test Code Flakiness in Mobile Apps: The Developer's Perspective

Valeria Pontillo[a,b], Fabio Palomba[a], Filomena Ferrucci[a]

[a] *Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy*
fpalomba@unisa.it, fferrucci@unisa.it
[b] *Software Languages (Soft) Lab — Vrije Universiteit Brussel, Belgium*
valeria.pontillo@vub.be

Abstract

Context: Test flakiness arises when test cases have a non-deterministic, intermittent behavior that leads them to either pass or fail when run against the same code. While researchers have been contributing to the detection, classification, and removal of flaky tests with several empirical studies and automated techniques, little is known about how the problem of test flakiness arises in mobile applications.

Objective: We point out a lack of knowledge on: (1) The prominence and harmfulness of the problem; (2) The most frequent root causes inducing flakiness; and (3) The strategies applied by practitioners to deal with it in practice. An improved understanding of these matters may lead the software engineering research community to assess the need for tailoring existing instruments to the mobile context or for brand-new approaches that focus on the peculiarities identified.

Method: We address this gap of knowledge by means of an empirical study into the mobile developer's perception of test flakiness. We first perform a systematic grey literature review to elicit how developers discuss and deal with the problem of test flakiness in the wild. Then, we complement the systematic review through a survey study that involves 130 mobile developers and that aims at analyzing their experience on the matter.

Result: The results of the grey literature review indicate that developers are often concerned with flakiness connected to user interface elements. In addition, our survey study reveals that flaky tests are perceived as critical by mobile developers, who pointed out major production code- and source code design-related root causes of flakiness, other than the long-term effects of recurrent flaky tests. Furthermore, our study lets the diagnosing and fixing processes currently adopted by developers and their limitations emerge.

Conclusion: We conclude by distilling lessons learned, implications, and future research directions.

Keywords: Test Code Flakiness; Software Testing; Mobile Apps Development; Mixed-Method Research.

1. Introduction

The increasing world digitalization, the changes we are experiencing in terms of communication, and the long-term effects of the era of social distancing are just some of the reasons why people need more and more mobile applications that let them connect and support their daily activities [59]. It is indeed not surprising to see

that we currently have more mobile devices than people.[1] In such a context, the quality of mobile applications is crucial to let developers create sustainable, successful, and dependable apps that can stay in the market and keep acquiring users [73, 84]. For instance, research has shown that internal properties of mobile software, like energy consumption [21], presence of design flaws [69], or even faulty APIs [9, 101], might impact the user's satisfaction and commercial success of mobile applications [102, 121]. Among the various methods to control for source code quality (e.g., design-by-contract [16] or formal methods [51]), software testing is one of the most effective practices to verify that the behavior of a mobile app meets the expectations and that non-functional attributes are preserved [70, 90]. The relevance of software testing is even more evident in the context of mobile applications development, where continuous improvements and releases represent a threat to software reliability [81].

Unfortunately, recent research has highlighted several limitations that make software testing still poorly applied by mobile developers, like the lack of practical automated testing tools [54] or the low attitude of developers to write unit tests [90]. In addition, the poor quality of test cases [49] might lead developers not to rely anymore on the outcome of the testing activities and even lead them to ignore actual defects in production code [106, 113].

One of the most recurring issues connected to test code quality is *test flakiness* [72]. This problem arises when a test case has a non-deterministic behavior, i.e., it may either pass or fail when run against the same code [26, 65, 72]. Over the last years, flaky tests have been deeply investigated from different perspectives [86]: researchers attempted to classify the root causes making a test flaky [26, 62, 72, 125], their lifecycle [40, 62], and their reproducibility [64]. At the same time, notable advances have been carried out in terms of automated detection [11, 12, 41, 109], classification [4, 45, 92, 93], root cause identification [61, 63, 125] and removal [42, 103].

While most of these studies targeted traditional software, our research points out a worrisome lack of knowledge on how test flakiness manifests itself in the context of mobile applications, how developers deal with it, and how mobile-specific (semi-)automated support might be designed. More specifically, the current state of the art has studied mobile flakiness only *tangentially* or *partially*. On the one hand, Gruber and Fraser [38] and Habchi et al. [43] surveyed practitioners to elicit their perspective on various properties of the problem, including how practitioners define the problem, the root causes of flakiness, and their reactions to the emergence of flaky tests. These studies targeted the generic population of developers, hence tangentially considering mobile developers as well. Nonetheless, their findings did not aim at identifying the peculiarities of test flakiness in mobile apps, but rather to provide general observations on the problem. In addition, neither Gruber and Fraser [38] nor Habchi et al. [43] analyzed the specific identification and mitigation processes put in place by mobile developers to deal with flakiness. On the other hand, Thorve et al. [110] proposed the first investigation into test flakiness in ANDROID apps: the authors replicated the seminal work by Luo et al. [72], classifying the root causes of flakiness through manual analysis of flakiness-inducing commits. The findings of the study showed that mobile apps have unique characteristics and, indeed, they present flaky tests whose root causes are different from those previously

[1] Smartphone users worldwide: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

investigated in traditional software, e.g., bugs related to *Program Logic* and the *User Interface*. The scope of the work by Thorve et al. [110] was limited to the analysis of the root causes of flakiness, yet it represented a call for further and more comprehensive research on the matter.

> **◎ Objective of the work.**
>
> *Our work builds on top of these previous findings and advances the state of the art by conducting a targeted and comprehensive analysis of how practitioners deal with test flakiness in mobile applications. Specifically, we aim to understand (1) How prominent and problematic flaky tests are; (2) What are the common root causes of test flakiness and how hard it is for developers to diagnose and fix them; and (3) What are the diagnosing and fixing activities that developers adopt when dealing with flaky tests.*

Through our analysis, we aim at providing the software engineering research community with insights that may be helpful in designing novel instruments to support practitioners or to tailor existing ones to fit the peculiarities of mobile applications. We approach our research by means of mixed-method research that combines two complementary studies. First, we conduct a systematic grey literature review that explores the developer's discussions and perspectives about test flakiness. Second, we extend the previous analysis through a survey study that involves 130 experienced mobile developers who were inquired about the state of the practice on test flakiness in mobile development. The choice of the research method was motivated by the recent findings by Zolfaghari et al. [126]: the authors showed that, despite the advances made over the last decade, the academic research community is still behind industries in terms of test flakiness management. As such, we address our objectives by focusing on the practitioners' knowledge through two complementary research instruments that may contribute to understanding the current state of the practice and the limitations thereof.

The results of both studies converge toward the same conclusions. We show that test flakiness represents a key problem for mobile developers as well, who would like to have further support to deal with it. This is especially true for flaky tests connected to the user interface. In addition, we elicit the typical processes that developers conduct when detecting, diagnosing, and fixing flaky tests. Last but not least, our research lets emerge that third-party APIs and production code-related factors might contribute to test flakiness.

Based on these findings, we conclude our paper by distilling a number of lessons learned, implications, and novel research avenues that researchers might consider to further build on the current state of the art.

Structure of the paper. Section 2 discusses the literature on flaky tests. Section 3 overviews our empirical setting. Section 4 reports the design and results of the systematic grey literature review, while Section 5 elaborates on the survey and the related results. In Section 6 we further discuss on the implications that our findings have for both researchers and practitioners. Section 7 summarizes the threats to validity of our study and how we mitigated them. Finally, Section 8 concludes the paper and outlines our future research agenda on the matter.

2. Related Work

The problem of flakiness is widely recognized and discussed by industrial practitioners worldwide: dozens of daily discussions are opened on the topic on social networks and blogs (e.g., [31, 75]). Researchers have started investigating the problem from multiple perspectives pushed by such an industrial movement. Systematic analyses of state of the art were presented by Barboni et al. [8], Parry et al. [86], and Zheng et al. [124]. At the same time, to the best of our knowledge, there is still no systematic analysis of the grey literature [35]. This point already motivates our work, which proposes the first systematic grey literature on test flakiness and the first exploration of specific aspects connected to test flakiness in mobile applications discussed by practitioners in the grey literature.

In the following, we discuss the literature on test flakiness by discussing (1) the main scientific activities conducted on the matter in the context of traditional software systems; (2) the papers that targeted the developer's perception of flaky tests; and (3) the currently available literature on test flakiness in mobile apps.

2.1. Test Flakiness in the Wild

In terms of scientific research, test flakiness has been widely studied in the context of large, open-source traditional systems, and, in the last few years, several automated techniques have been proposed [86]. These approaches cover various angles of the flakiness problem, such as the detection of flaky tests [11, 12, 20, 37, 41, 44, 67, 77, 88, 109, 118], the negative effects that flaky tests may create during regression testing [74], e.g., missing deadlines because certain features cannot be tested sufficiently, the prediction of test flakiness [4, 28, 39, 45, 92–94, 115], the identification of the root cause of flakiness [47, 61, 63, 76, 125] and their removal [42, 47, 68, 103], using various algorithms (e.g., machine learning or search-based) and methods (e.g., static versus dynamic analysis).

Interestingly, researchers and practitioners have also been working together to investigate test flakiness. Indeed, a growing number of industrial studies have been carried out that report on empirical investigations or propose tools. Lampel et al. [66] proposed a new approach that automatically classifies failing jobs as pertaining to software bugs or flaky tests. Rehman et al. [97] quantified how often a test fails without finding any defect in production code through an empirical investigation across four large projects at ERICSSON. From an empirical software engineering viewpoint, Luo et al. [72] manually inspected 1,129 commits and reported a set of ten root causes of test flakiness. Lam et al. [61] also conducted a large-scale study of flakiness in MICROSOFT, emphasizing the problematic nature of flaky tests.

2.2. Developer's Perception of Test Flakiness

Multiple investigations have been previously conducted to understand the developer's perception of test flakiness. These studies are clearly closely related with respect to the work we propose. Table 1 summarizes the key articles in this area along with the main differences between them and our study.

Eck et al. [26] surveyed 21 MOZILLA developers in order to classify the root causes of the flaky tests they previously fixed. While the authors confirmed most of the root causes identified by Luo et al. [72], they also identified

Table 1: Comparison between our study and the most closely related papers.

Related Work	Main Focus	Differences
Eck et al. [26]	A mixed-method study to understand the developer's perspective on test flakiness. First, 21 professional developers classified 200 flaky tests previously fixed, in terms of the nature and the origin of the flakiness. Then, the authors conducted an online survey with 121 developers with a median industrial programming experience of five years - a multivocal literature review was conducted to inform the survey with the challenges faced by developers when dealing with flaky tests.	• No focus on mobile applications; • The context of the study was on industrial practitioners; • The goal of the multivocal literature review was to identify the challenges faced by developers when dealing with flakiness rather than to collect the practices employed by developers.
Habchi et al. [43]	A qualitative study with 14 practitioners to identify the sources and the impact of flakiness, the measures adopted by practitioners and the automation solutions. A grey literature review was conducted to inform the design of the semi-structured interviews.	• No focus on mobile applications; • The study was based on the experience on 14 people, while our work is based on a large amount of mobile practitioners; • The grey literature review was exploratory and non-exhaustive.
Gruber and Fraser [38]	An empirical study in which the authors surveyed 335 professional software developers and testers in different domains to understand the prevalence of test flakiness, how flaky tests affect developers and what developers expect from researchers.	• The target audience and the participant selection are different; • The work had a specific focus on the support that developers need from the tools; • The work did not conduct a systematic literature review.
Ahmad et al. [2]	A multiple-case study to understand flakiness in a closed-source development context. The study showed 19 factors categorized as *test code*, *system under test*, *CI/test infrastructure*, and *organization-related* that are directly related to test flakiness.	• The study was based on the experience on 18 developers in a closed-source context, while our work is based on a larger number of practitioners; • The work was focused on the root causes of flaky tests and the practitioner's perception, while our study also investigates the strategies applied by practitioners to deal with flakiness in mobile apps; • The work did not conduct a systematic literature review.
Parry et al. [87]	A mixed-method study to understand how developers define flaky tests, their experiences on the impact and causes of test flakiness, and the actions taken in response to flaky tests. The authors deployed a survey that obtained 170 responses and analyzed 38 STACK OVERFLOW threads related to test flakiness.	• No focus on mobile applications; • Our work includes the investigation of the most frequent root causes and the problematic nature of test flakiness in mobile apps; • The work did not conduct a systematic literature review.

additional implementation issues inducing flakiness. It is worth remarking that Eck et al. [26] also conducted a follow-up survey study to analyze the developer's perception of test flakiness - this survey was informed by a multivocal literature review targeting the challenges faced by practitioners when dealing with test flakiness. By collecting 121 responses, the authors found that flaky tests are perceived as relevant by the vast majority of the traditional developers involved. In addition, they found that the emergence of flakiness-related concerns might

impact socio-technical activities like resource allocation and scheduling. This study has multiple differences with respect to ours. First, we design the study to investigate how flakiness manifests itself in mobile applications: as such, our results on the root causes of flakiness should be seen as complementary to those of Eck et al. [26], who analyzed an industrial case such as the one of MOZILLA. Second, the authors conducted a multivocal literature review and a survey study. Nonetheless, the scope of their analyses is diametrically different with respect to ours. They indeed conducted a review with the sole goal of identifying the challenges faced by practitioners when dealing with flakiness and validated the identified challenges with a survey study. On the contrary, we are interested in understanding the practices adopted by practitioners when identifying, diagnosing, and fixing strategies. In this sense, our study provides a wider overview of how mobile developers face test flakiness.

Habchi et al. [43] built upon the findings by Eck et al. [26] and conducted an interview study involving 14 industrial practitioners. Their results confirmed the relevance of the problem but also pointed out that in a non-negligible amount of time, flakiness stems from interactions between the system components, the testing infrastructure, and other external factors. With respect to this paper, our study approaches the problem with a different research method, i.e., a survey study versus an interview study. This difference allowed us to reach more developers than Habchi et al. [43] (150 versus 14), hence reinforcing the ecological validity of our findings.

It is worth mentioning that Habchi et al. [43] did not explicitly focus on the problem of flakiness in mobile applications, hence considering a more general set of resources that might have provided information on how developers deal with flaky tests. On the contrary, our goal was to investigate the perspective of mobile developers and, for this reason, we tailored the systematic grey literature process to mobile apps, thus eliciting and analyzing information that are specific to the mobile software development context.

In the second place, Habchi et al. [43] focused on mapping the measures employed by practitioners when dealing with flaky tests. Our work is, instead, larger and more comprehensive, as we aimed to analyze (1) the prominence, (2) the root causes, and (3) the diagnosis and mitigation strategies applied by mobile developers when dealing with flakiness.

Finally, the grey literature developed by Habchi et al. [43] did not comprehensively survey the resources available but aimed to inform the design of semi-structured interviews—according to the authors, they performed a literature review which was *"exploratory, not exhaustive, and only aimed at informing the design of the semi-structured interviews"* [43]. In other terms, their work did not systematically follow the guidelines for conducting these types of studies; on the contrary, we performed multiple steps to make our literature review as complete as possible, discarding too generic or unreliable resources that might have biased our conclusions.

Gruber and Fraser [38] surveyed 335 professional software developers and testers in different domains; their results confirmed the relevance of the problem, especially when using automated testing. While the authors involved practitioners working on various domains, including the development of mobile applications, their focus was on the automated support needed to analyze test flakiness. As such, our survey is complementary and aims at enlarging the knowledge of how mobile developers identify, diagnose, and fix flaky tests. In addition, the grey

literature review represents an additional contribution we provide.

Ahmad et al. [2] interviewed 18 industrial developers on the root causes of flakiness and their perception of the harmfulness of flaky tests. On the one hand, our work is different since it targets mobile developers. On the other hand, our scope is larger as we aim to understand the practices used by practitioners to deal with flakiness.

Finally, the empirical study conducted by Parry et al. [87] involved practitioners in analyzing their definition of test flakiness, their experiences, and the actions taken in response to newly identified flaky tests. While the goals of this paper are somehow similar to ours, we explicitly target mobile application developers to identify the specific actions conducted by those developers while addressing the problem of flaky tests. In addition, we also elaborate on the most frequent root causes of flakiness and the problematic nature of test flakiness as perceived by developers. Finally, Parry et al. [87] conducted the study by combining the survey study with the analysis of StackOverflow posts. On the contrary, we conduct a grey literature review to address the research questions of the study. Therefore, also in this case, our study can be seen as complementary to the one of Parry et al. [87].

On the basis of the analysis just discussed, our contribution compared to the related work presented in Table 1 (1) is, therefore, the first exhaustive grey literature review on test flakiness in the context of mobile applications, (2) proposes the first survey study that explicitly involves mobile developers inquiring them on the peculiarities of flakiness in mobile apps, and (3) extends the current body of knowledge through a complementary analysis on how mobile developers identify, diagnose, and fix flaky tests.

2.3. Test Flakiness in Mobile Applications

The population-level differences between mobile apps and non-mobile apps have been the subject of a number of previous works. For instance, the seminal work by Wasserman [117] pointed out several aspects making mobile applications different and, to some extent, special: the potential interaction that a mobile app may have with other applications and services, the need for handling sensors, the user interface which needs to adhere to externally developed guidelines, other than the different weight that some non-functional attributes, e.g., energy consumption, may have on how the app is developed are just some of the elements that naturally make mobile applications different from other applications. In addition, it is worth considering that mobile apps can be native or hybrid - this further influences the way the app might interact with the telephone network or the internet and the way the app should acquire and display data on the device. Furthermore, software engineering researchers have also found out that the development process behind the creation of mobile apps is different from the traditional one (e.g., [33, 52, 123]), impacting in different manners the way these apps are developed and tested.

Regarding test flakiness, the key aspect to consider is that mobile apps run on a large variety of devices with different technical characteristics and display sizes [29]. This makes mobile apps more sensitive to various problems. Thorve et al. [110] investigated the nature of flakiness in Android using a research methodology inspired by the work by Luo et al. [72]. The authors found various root causes which are unique to mobile apps, like *"Program Logic"* and *"UI"*. The former refers to the flakiness induced by wrong assumptions about the app's program be-

havior, while the latter refers to flakiness due to the design and rendering of widget layouts on user interfaces. The need to run on multiple devices may cause additional issues with test flakiness due to the dependency between the app and the underlying platform it is running on. Indeed, Thorve et al. [110] discovered a root cause, coined *"Dependency"*, which concerns the flakiness caused by the usage of specific hardware, OS version, or a third-party library. At the same time, other root causes frequently reported in previous studies targeting non-mobile applications are less common or even irrelevant in mobile apps. This is the case, for instance, of the root causes known as *"Test Order Dependency"* and *"Resource Leak"*, which were never discovered in the analysis of Thorve et al. [110].

The observations above indicate the existence of peculiarities that make mobile apps different from other systems and further motivate the need to investigate the problem of test flakiness in such a specific context - which is indeed the ultimate goal of our empirical investigation.

Other relevant papers that elaborated on test flakiness in mobile apps are those by Dong et al. [23], Romano et al. [99], and Silva et al. [105]. In particular, Dong et al. [23] proposed FLAKESHOVEL, a tool for detecting ANDROID-specific flaky tests through the systematic exploration of event orders. Romano et al. [99] analyzed 235 flaky UI tests to understand common root causes, the strategies used to identify the flaky behavior and the fixing strategies proposed. Finally, Silva et al. [105] proposed SHAKER, a lightweight technique to improve the ability of the RERUN approach to detect flaky tests. Their study provided guidelines for creating reliable and stable UI test suites.

Despite the advances made by researchers on test code flakiness, there is still a lack of knowledge on the problems and processes that developers face when dealing with flaky tests in mobile apps. Our study fills this gap and provides a starting point for future research to improve mobile app development and testing practices.

3. Research Questions and Settings

The *goal* of the empirical study was to collect and analyze the mobile app developer's perspective on test flakiness with the *purpose* of eliciting (1) the prominence of the problem, (2) the most common root causes, and (3) the strategies implemented by developers to diagnose and fix flaky tests. The *perspective* is of both researchers and practitioners. The former are interested in assessing the current state of the practice, hence understanding how to further support mobile developers when dealing with flaky tests. The latter are interested in understanding how other mobile developers deal with the problem, thus learning potential strategies to put in place.

More specifically, our study was driven by three main research questions. In the first place, we aimed at collecting the developer's perception concerning the prominence and problematic nature of test flakiness. Such an investigation allowed us to challenge the results obtained in traditional contexts by Eck et al. [26], hence complementing their observations with data coming from mobile application developers. In particular, we asked the following research question:

RQ$_1$. *How prominent and problematic is test flakiness as perceived by mobile app developers?*

Once we had assessed the harmfulness of the problem, we moved toward understanding of the most common causes of test flakiness and the effort spent when dealing with them. This analysis first allowed us to complement the many studies that previously targeted the identification of the root causes of flakiness [26, 61, 62, 72, 125]: we were indeed interested in providing insights into the root causes that might be more prominent in mobile apps. Secondly, this research question also had the goal of challenging the findings reported by Thorve et al. [110], who investigated the root causes of test flakiness in mobile apps. We asked:

RQ$_2$. *What are the most common causes of test flakiness and how hard is for developers to deal with them?*

Finally, we focused on eliciting the strategies performed by mobile app developers when it turns to diagnosing and fixing flaky tests. This analysis allowed us to delineate the test flakiness problem from the process perspective, hence providing insights into how developers practically deal with test flakiness—this might be particularly relevant for researchers interested in supporting developers during those processes. We, therefore, asked:

RQ$_3$. *How do mobile app developers approach flakiness in terms of diagnosing and fixing strategies?*

To address our research questions, we proceeded with two complementary investigations. On the one hand, a systematic grey literature review aims at analyzing how mobile app developers discuss the problem and which strategies they employ to identify, diagnose, and remove flaky tests. On the other hand, a survey study aimed to corroborate and further extend the findings obtained through the systematic grey literature review, possibly providing additional insights from the direct inquiry of practitioners involved in the flaky test management process.

The conclusions drawn at the end of the empirical study aimed to contribute lessons learned, implications, and open challenges on the practices and issues that developers currently face when dealing with flaky tests. By definition, the empirical study has an *exploratory* nature, as its goal is to elicit the developer's opinions about a problem. In terms of design, reporting, and presentation, we followed the empirical software engineering guidelines by Wohlin et al. [120], other than the *ACM/SIGSOFT Empirical Standards*.[2]

4. Analyzing the Grey Literature

The first step of our analysis consisted of designing and executing a systematic grey literature review. The choice to conduct a systematic grey literature review has several reasons: (1) to increase the likelihood of a comprehensive search [15], (2) to avoid publication bias [122], and (3) to allow the analysis of the point of view of practitioners [122]. We followed well-established research guidelines to conduct it [35], as elaborated in the remainder of the section.

[2]Available at: `https://github.com/acmsigsoft/EmpiricalStandards`. Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"*, *"Systematic Reviews"*, and *"Qualitative Surveys"* guidelines. All the material collected and employed to address the research questions have been anonymized and made publicly available in the online appendix [95].

4.1. Data Sources and Search Strategy

Similarly to previous work in literature [43, 60], we employed the GOOGLE search engine to search for the grey literature. It is important to note that Garousi et al. [35] defined a model that describes the grey literature's layers (or "shades"). The first layer concerns books, magazines, government reports, and white papers, which are considered as high credibility sources. The second layer is annual reports, news articles, presentations, videos, Q/A sites, and Wiki articles, which are considered as moderate credibility sources. The third layer consists instead of blogs, emails, and tweets, which are deemed to have lower credibility. In an effort to systematically analyze all the resources available on test flakiness in mobile development, we decided to include all the layers of grey literature, hence also including the risky sources coming from the third layer. Yet, as explained in Section 4.2, we accounted for this aspect by means of additional quality checks aiming at assessing the credibility of the sources included in our systematic analysis.

As for the search query, which represents the set of keywords to search sources on the phenomenon of interest, we designed it to account for the multiple terms/synonyms that could be used to refer to test flakiness, e.g., *intermittent* or *non-deterministic* test. Hence, we set the initial search terms and created the following query:

> **Search Query.**
>
> *("flaky test*"* OR *"flakiness"* OR *"intermittent test*"* OR *"non-deterministic test*"* OR *"nondeterministic test*"* OR *"nondeterminism")* AND *("mobile"* OR *"Android"* OR *"iOS")*

We applied the query on GOOGLE, scanning each resulting page until saturation, i.e., we stopped our search when no relevant resources emerged from the results, as suggested by Garousi et al. [35]. We performed our search in an incognito mode to avoid our personal search bias. As recommended by Garousi et al. [35], we did our best to exclude resources providing outdated information that might have biased the conclusions drawn in the study. In particular, we excluded all the resources published before 2018: the median lifespan of a mobile operating system version is 4.6 years [7, 19] and, therefore, it is likely that testing frameworks and concerns present in older versions are not anymore relevant nowadays. In other terms, we preferred to include only the most relevant resources in an effort to present the current state of the practice.

4.2. Eligibility Criteria and Quality Assessment

On the basis of the resources identified from the search, we then verified the eligibility criteria and performed a quality assessment. These were required to select the most appropriate and relevant resources to synthesize.

Inclusion/Exclusion Criteria. Inclusion and exclusion criteria allow the selection of resources that address the goal of our literature review [35]. In the context of our study, we applied the following criteria.

A) **Inclusion criteria**: Resources that analyze test flakiness in mobile applications from different perspectives, i.e., root causes, diagnosing and fixing strategies were *included* in our study.

B) **Exclusion criteria**: The resources that met the following constraints were filtered out from our study:

- Resources not written in English;
- Resources that do not match the focus of the study;
- Resources that are restricted with a paywall;
- Duplicated resources;
- Resources that analyze flakiness from the perspective of academics.

Quality Assessment: Before proceeding with the extraction of the data from our resources, we assessed the credibility, quality, and thoroughness of the retrieved resources to discard the results that did not provide enough details to be used in our study. Starting from the quality assessment checklist presented by Garousi et al. [35], we defined a checklist that included the following questions:

Q1. *Is the publication organization reputable?*

Q2. *Is the author known?*

Q3. *Does the author have expertise in the area?*

Q4. *Does the source have a clearly stated purpose?*

Each question could be answered as *"Yes"*, *"Partially"*, and *"No"*. We associated a numeric value for each label to better assess the quality and thoroughness of each source: the label *"Yes"* was associated to the value *'1'*, *"Partially"* to *'0.5'*, while the label *"No"* was associated to *'0'*. The overall quality score was computed by summing up the score of the answers to the four questions, and the articles with a quality score of at least 2 were accepted.

The quality assessment was mainly carried out by the first two authors of the paper, who manually dived into the resources identified in order to address the questions in the checklist. More specifically, the inspectors applied the following steps to address each of the questions on the checklist:

- **Q1.** The inspectors took the name of the organization mentioned in the resource as input. If the organization was publicly known and considered reputable by the inspectors, e.g., SLACK,[3] it was considered reputable. If the organization was unknown to the inspectors, the inspectors performed a search over internet to evaluate the reputation of the organization by looking at its mission, i.e., whether the organization was connected to software programming and testing, its social presence, i.e., whether the organization had a LINKEDIN account, the number of followers of the social accounts, and any additional contextual information that

[3] SLACK: https://slack.com/

might have led the inspectors assess its reputation, e.g., size of the organization, number of branches of the company and their geo-dispersion. Upon the collection of these data, the inspectors opened a discussion and finally provided an assessment of the reputation of the organization. We did not set any threshold on the pieces of information gathered, but we used such pieces of information to have a contextual understanding that might have allowed us to better assess the reputation of the organizations and authors. The use of fixed thresholds could have biased our evaluation or filtered out relevant resources: as such, we preferred to conduct a qualitative assessment of the resources by discussing the organization rather than relying on quantitative measures.

- **Q2-Q3.** To assess the author of a post and their related expertise, the inspectors conducted a similar process as described for **Q1**. First, they verified that the post had an author. Then, they took the name of the author as input and snowballed the search over the internet with the aim of gathering information and evaluate the credibility of the author by looking at their previous/current positions, social media presence, and its skills in terms of software testing according to the information available on the internet, e.g., personal website or company pages. If the inspectors were able to link a resource to an author with some expertise on the matter, they considered the author reputable and the resource worthy to be analyzed.

- **Q4**. A resource was considered to have a clearly stated purpose if it explicitly reported information and/or points of view on the problem of test flakiness. In other terms, the inspectors verified that the content of the resource was specific enough to address the research questions of the study.

The material was equally distributed between the inspectors. In problematic cases of filtration, the inspectors opened a discussion revolving around the parameters described above and found an agreement on whether a resource should have been included or not. In case of disagreement, the third author was involved and the decision was taken based on majority voting. This did not eventually happen, as the discussions between the first two inspectors already addressed the source selection process.

4.3. Execution of the Grey Literature Review

Once defined the steps for our grey literature review, we proceeded with its execution. Figure 1 overviews the process, reporting the inputs/outputs of each stage as well as summarizing the number of resources considered for our study. The entire process was executed in the period between November 15 to April 30, 2023, and took around 70 person/hour. The potentially relevant results for our search query were 187, but after an initial pre-screening—where we discarded resources not related to the topic, articles belonging to white literature, Wikipedia entries, or repeated entries—we filtered out 77 sources, obtaining a final list of 110 relevant sources. Next, the inspectors applied the Inclusion/Exclusion criteria. This step led to filtering out 87 sources, for a final list of 23 sources. The last step of our process was the quality assessment, where six additional resources were discarded. Table 2 reports

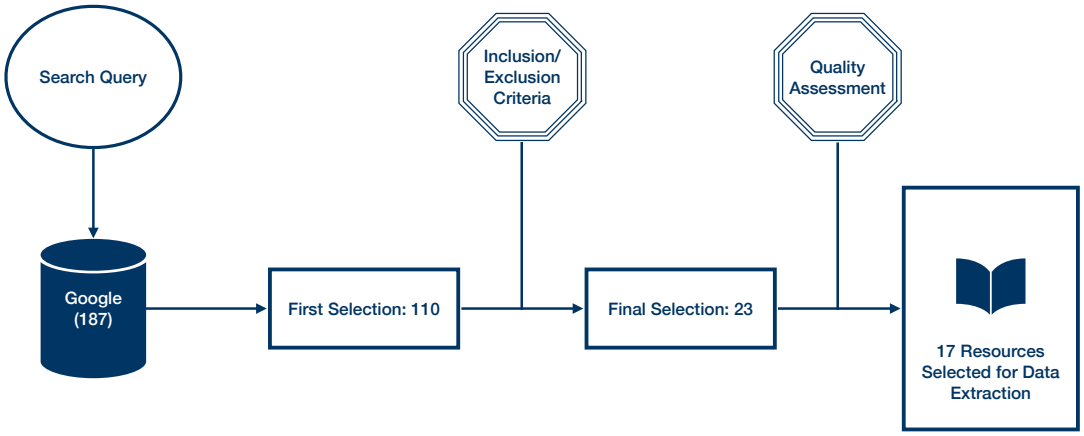

Figure 1: Overview of the resources selection process.

the outcome of the quality assessment step: More particularly, for each resource excluded as a result of the quality assessment, the table overviews (i) the description of the resource; (ii) the motivation leading to the exclusion; and (iii) the score assigned to each quality criterion used for the assessment. As a conclusion of the selection process, we proceeded with a set of 17 online resources of different types, e.g., articles, blog posts, useful for our grey literature review. The relatively low amount of resources identified does not necessarily influence the prominence of flaky tests perceived by mobile developers. On the one hand, all the resources selected as part of our grey literature review come from organizations deemed reputable, hence suggesting that the problem of flakiness is indeed serious in multiple contexts. On the other hand, test flakiness is a technical matter typically discussed by developers with strong experience in software testing, which naturally limits the amount of resources available.

Moreover, we are aware that such a relatively low amount of resources left after the application of the quality assessment may lower the generalizability of the conclusions drawn from our analysis. In this respect, two observations can be made. First, we aimed at performing an extensive analysis of the grey literature on flaky tests in mobile applications. While more resources were available when considering the problem from a general perspective, i.e., when including all the articles that discuss flaky tests independently from the domain, only a limited amount of resources were valid when it came to the mobile application domain. As such, our analysis simply overviews the current state, highlighting the scarcity of resources available. Second, and perhaps more importantly, the few results obtained motivated the follow-up survey study, which was designed to complement and extend the findings obtained from the grey literature review. In our online appendix [95], we made available a file that contains both the resources included in the analysis and those excluded at each step of the reviewing process.

Upon completion of the quality assessment, we then proceeded with the data extraction and analysis. In particular, the first two authors of the paper went through each resource, reading and summarizing its content. For each of them, they also assigned a label describing the main theme treated in the resource, e.g., a source report-

Table 2: The outcome of the quality assessment with a brief description of the content, the motivation behind the exclusion and the score assigned to each quality criteria, i.e., reputable organization, author known, expertise on the topic and a clearly stated purpose.

Resource	Description	Motivation	Q1	Q2	Q3	Q4
RE1	The resource cites test flakiness as a challenge when implementing mobile test automation.	The author and his expertise are not known. The organization is known (REPEATO is a test automation tool that works based on computer vision and machine learning), but the resource does not provide additional information useful to answer our research questions.	1	0	0	0.5
RE2	The resource shows an Android mono repo case study to understand the unreasonable effectiveness of the test.	The author's name is reported but it has been impossible to find more information about his expertise; the article is written in a personal blog. For this reason, it is impossible to analyze the expertise.	1	0	0	0
RE3	The resource presents a generic description of test flakiness and how to reduce it. The focus is not on mobile apps.	The author and his expertise are not known. The organization is known (TESTINIUM), but its mission is unclear. Does the organization provide a framework, a specialized testing team, or other?	0.5	0	0	1
RE4	The resource presents a test automation tool, i.e., DETOX and reports the added advantages of being less flaky without explaining how.	SPRITECLOUD is a community of software quality assurance and cybersecurity testers in Amsterdam, but it is a sort of blog. The author and his expertise are unknown and the resource just mentions flakiness. It is impossible to evaluate the criteria related to the author and the organization.	0	0	0	0.5
RE5	The resource provides an overview of flakiness and how to fix it.	The author is unknown (the author used a nickname) and the article is written in a personal blog. For this reason, it is impossible to analyze the expertise.	0	0	0	1
RE6	The resource provides an overview of the top mobile app testing frameworks for 2022 and names flakiness about a tool (XCUITEST) to say that it minimizes it.	The author is unknown (the author used a nickname) and the article is in a personal blog. For this reason, it is impossible to analyze the expertise.	0	0	0	1

ing on *"Mitigation Strategies"*, and a brief summary of the content. These pieces of information allowed the two authors to describe and categorize the content of the resources. These were later jointly discussed in order to homogenize the labels and descriptions. The discussion lasted 4 hours and the resulting coding exercise was used to analyze the results, as reported in Section 4.4.

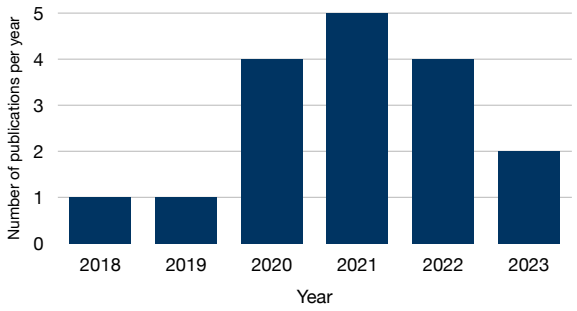

Figure 2: Number of resources by publication year. This grey literature review was conducted part-way through 2023. Hence for this year, not all resources were considered.

4.4. Analysis of the Results

Before discussing the main findings coming from the systematic grey literature review, let us discuss the years of publication of the resources included in the study. Figure 2 shows the distribution of the number of resources over the years. From the figure, we may notice again that only a few developers discuss the problem of flakiness in

mobile apps over the internet. At the same time, we observed an increasing trend, with the number of articles that were raised in 2021. The few developers discussing the problem might be more aware of the problem, possibly presenting interesting insights into the root causes and processes adopted to deal with flaky tests.

Table 3: Final resource of the Grey Literature Review with a brief description of the content, the publication year, and at least one label describing the main theme treated in the resource.

Resource	Description	Year	Label
R1	Strategies to reduce flakiness in ESPRESSO UI Tests (e.g., disable animation).	2020	Mitigation Strategy
R2	Explanation of a workflow to reduce flakiness.	2021	Detection Strategy
R3	The Mobile Developer Experience Team, DevXP, describes the path to minimize test flakiness.	2022	Detection Strategy, Mitigation Strategy
R4	Lecture where the speaker suggests some best practices for increasing test stability are learned. The speaker suggests that Android developers often use the word *flakiness*.	2022	Developers' perspective
R5	Description of some root causes of flakiness in mobile apps (e.g., communication with external resources) and some strategies to fix flaky tests.	2021	Root Cause, Fixing Strategy
R6	Description of APPIUM and some tips to avoid test flakiness.	2018	Mitigation Strategy
R7	Overview of some root causes of test flakiness and how to avoid them.	2021	Root Cause, Mitigation Strategy
R8	Tutorial on facilitating the User Interface testing and reducing test flakiness.	2020	Mitigation Strategy
R9	Presentation of some strategies in XCODE to detect test flakiness, i.e., the test repetition mode.	2022	Detection Strategy
R10	The author describes how to mitigate test flakiness in the emulator.	2021	Mitigation Strategy
R11	Description of GORDON, an ANDROID plugin that makes a report on test flakiness.	2019	Detection Strategy
R12	The author shows an example of a test case that could have some problems related to non-determinism. Then, the author shows how to avoid this problem.	2021	Mitigation Strategy
R13	Presentation of BITRISE[4], a platform to build mobile applications that can detect flakiness by analyzing the result from various builds and reporting which tests change behavior.	2023	Detection Strategy
R14	Android Documentation on UI Test. The resource suggests avoiding the use of the `sleep()` function because this makes tests unnecessarily slow or flaky because running the same test in different environments might need more or less time to execute.	2022	Mitigation Strategy
R15	Since 2017, in SLACK there is the Mobile Developer Experience (DevXP) team, that focused on several areas of mobile application development, including *"Automation test infrastructure and automated test flakiness"*. The team analyzed flakiness in the last years and developed an approach to reduce flakiness from 57% to 4% (the approach is described in R3).	2022	Detection Strategy
R16	A simple list of best Android testing tools. There is a reference to flakiness detection for Waldo, a no-code testing tool useful for teams that do not have dedicated developers. The tool analyzes the change between the various builds to detect a flaky test.	2023	Detection Strategy
R17	This master thesis project gives guidance to mobile application developers and testers when planning to execute automated tests. The project performed a comparison of the three most popular mobile application testing automation in Agile/DevOps environments and used the flakiness as a measure to understand the degree of the frameworks to handle flaky tests based on the number of `Wait` statement declared.	2020	Mitigation Strategy

The results of the systematic grey literature review are presented in Table 3. Specifically, for each resource included in the study, the table reports the brief summary and label assigned during the data extraction and analysis phase. More details about the authors and the URL are reported in the online appendix [95].

As the reader may observe, the 17 resources could be categorized into five categories: (1) *"Mitigation Strategy"*, (2) *"Detection Strategy"*, (3) *"Developer's perspective"*, (4) *"Root Causes analysis"*, and (5) *"Fixing strategy"*. It is worth noting that for R3, R5, and R7 we assigned two different labels, as they covered more themes. Discussions labeled as *"Mitigation Strategy"* and *"Detection Strategy"* were the most frequent (respectively nine and seven mentions),

indicating that mobile developers commenting on test flakiness were mainly concerned with how to detect test cases that could be potentially flaky and how to mitigate them. The least frequent theme was, instead, the one labeled as *"Developer's Perspective"* (1 mention). In the remainder of the section, we further discuss the results obtained, splitting the discussion by theme.

"Root Cause Analysis". Two of the twelve resources, i.e., R5 and R7, provided information about root causes. While in R5, we could find a detailed explanation of the most common root causes in IOS mobile applications, in R7, we could analyze a short list of the reasons behind test flakiness in ANDROID applications. In the former case, the author recognizes four common root causes such as *Race Conditions*, *Environment assumptions*, *Global State*, and *Communication with external services*. While some of the root causes mentioned are similar to those reported for traditional software, e.g., *race conditions*, additional factors seem to come into play in mobile apps. This is the case of *Global State* and *Communication with external services*. In particular, global states pertain to variables or instances globally available in the source code: mutable global configurations may be configured differently for different test classes, possibly affecting the execution environment and leading test cases to fail in certain scenarios. As for the external services, R5 reported that this is a common root cause of flakiness, which is mostly due to the strong reliance of mobile apps on third-party libraries and services.

In R7, the author recognizes four macro-categories as root causes of test flakiness. These macro-categories refer to problems connected to the design of production and test code, the use of a real device or an emulator, and the hardware infrastructure. Also in this case, we could recognize the existence of additional factors affecting the problem of test flakiness, like the design of production and test code—this is something that will be identified and discussed further in the survey study. Hence, the systematic work performed let emerge that test flakiness is not just a matter of test code when it comes to mobile applications, but rather further investigations into the root causes due to production code might be worthwhile. In addition, R7 highlights that flaky tests may be due to the specific test execution environment, i.e., the flakiness might arise when exercising the app against a real device rather than a simulator. The aspects highlighted in R5 and R7 let emerge new perspectives of the problem, which are peculiar to mobile apps.

Another interesting discussion point relates to the source code reported in Listing 1. It shows an example of UI flaky tests coming from R12. The listing shows how the UI events reliant on network calls can take different amounts of time. In fact, this test switches between two fragments in an activity, i.e., the click of the first button reveals the second button and the click on the second button reveals the first button again. As reported in R12, the `R.id.button_second` is not enabled until a network call is completed—this a *flaky test* because the test may attempt to access a button or a list item that is not available yet because it is awaiting a response from a test server. On the one hand, this example corroborates the results obtained by Romano et al. [99], who reported that the root causes behind UI flaky tests may be actually similar to those of traditional systems, e.g., they might be induced by network-related concerns. On the other hand, however, those flaky tests might be harder to manage

and fix, as multiple root causes might co-occur — this example highlights, once again, that the problem of flaky tests in mobile apps is peculiar and possibly requires more sophisticated instruments that might help mobile developers to diagnose and handle them.

```
@Test(expected=androidx.test.espresso.NoMatchingViewException::class)
    fun flakyTest() {
        onView(withId(R.id.button_first)).perform(click())
        onView(withId(R.id.button_second)).perform(click())
        onView(withId(R.id.button_first)).check(matches(isDisplayed()))
    }
```

Listing 1: Example of UI flaky tests in a mobile application. This test switches between two fragments in an activity and requires a network call to be completed. The awaiting response from a server could cause flakiness.

"Mitigation Strategy". The resources categorized as such were nine and all focused on the flakiness arising when testing the user interface of mobile apps. This result corroborates the findings by Thorve et al. [110], who showed that a large part of flaky tests in mobile applications is concerned with issues related to the user interface. More particularly, the authors of these resources offered an overview of how to reduce the risks of test flakiness when using the ESPRESSO testing framework. It is important to point out that, as reported by R14, *"Frameworks like* ESPRESSO *are designed with testing in mind, so there is a certain guarantee that the User Interface will be idle before the next test action or assertion"*. This suggests that in the last years, mobile developers are trying to put in place practices that can preemptively mitigate flakiness, similar to what happens with traditional software systems [63, 104, 116]. To clarify our analysis, Table 4 reports the mitigation strategies discussed in the resources.

Table 4: Mitigation Strategies with related resources.

Mitigation Strategy	Resources
Use the Wait Actions	R1, R6, R10, R12, R14, R17
Use the Locator Strategies	R6, R8, R10
Disable Animations	R1, R6
Mock the External Services	R6, R7
Isolate the Environment	R3, R7
Set the AppState	R6
Sort the test	R7

Some of the recommended strategies are not specific to mobile applications. For instance, the use of `WaitAction` functions to manage concurrency or the isolation of the testing environment are actions that might be performed independently from the mobile context. Other mitigation strategies are, instead, more connected to how mobile apps are designed. These pertain to disabling animations that might influence the outcome of a test, using locator strategies to identify the objects that might be problematic during the execution of the app, or setting an `AppState`, which consists of avoiding that multiple tests are executed at the same time. All the

strategies were contextualized for UI testing, which again points out the need for methods and techniques to support mobile developers when dealing with user interfaces. Besides reporting the common UI-related root causes, R12 also commented on the way they can be mitigated.

```
fun onViewEnabled(viewMatcher: Matcher<View>): ViewInteraction {
        val isEnabled: ()->Boolean = {
            var isDisplayed = false
            try {
                onView(viewMatcher).check(matches((ViewMatchers.isEnabled())))
                isDisplayed = true
            }
            catch (e: AssertionFailedError) { isDisplayed = false }
            isDisplayed
        }
        for (x in 0..9) {
            Thread.sleep(400)
            if (isEnabled()) {
                break
            }
        }
        return Espresso.onView(viewMatcher)
    }
```

Listing 2: A simple strategy to mitigate the flakiness previously shown.

For instance, Listing 2 shows how to manage the UI flaky test previously discussed. One simple solution is to use a view matcher that checks for the desired conditions before moving forward in the test. The `onViewEnabled` function checks whether the UI element being queried is enabled. If the element is not enabled, the function will use a small amount of time and check again. Once this function is introduced, the test case presented in Listing 1 will be modified as shown in Listing 3.

```
@Test
    fun idleViewMatcherTest() {
        onView(withId(R.id.button_first)).perform(click())
        onViewEnabled(withId(R.id.button_second)).perform(click())
        onView(withId(R.id.button_first)).check(matches(isDisplayed()))
    }
```

Listing 3: New UI test case to avoid non-deterministic behavior.

"Detection Strategy". Resources in this category refer to recommendations, tools, and frameworks that support developers in detecting test flakiness. We analyzed seven resources, i.e., R2, R3, R9, R11, R13, R15, and R16. The first resource refers to FLUTTER, an open-source GOOGLE framework for native and multi-platform applications. As reported by the author of R2, this framework provides an automatic detection strategy that scans the test execution statistics over the past 15 days.

R9 reports on a detection mechanism introduced within XCODE, an Apple IDE to develop software that supports several programming languages, e.g., C, C++, and JAVA. From version 13, the IDE made available the so-called *Test Repetitions Mode*, which is a method to re-run test cases multiple times in order to control for the presence of flaky tests. More particular, there are three types of test repetition modes currently supported in XCODE, i.e., *Fixed Iterations*, *Until Failure*, and *Retry on Failure* that can be used to detect flaky tests automatically.

R11 reports information about a GRADLE plugin for ANDROID applications. Once a test failure is detected, the plug-in, coined GORDON, tries to re-run the test in the same environment and in a different environment to define whether that failure depends on a bug or is a flaky test.

R13 presents BITRISE, a platform to build mobile applications. This platform recognizes a flaky test by analyzing the results from various build and reporting which test change behavior from one build version to another.

Looking at R16, this resource lists the best Android testing tools. In this list, we find WALDO,[5] a no-code testing tool useful for teams without dedicated developers. Also in this case, WALDO analyzes the behavioral changes between the various build version to detect the flakiness.

Perhaps more interesting is the detection strategy suggested in R3 and R15, which is concerned with the setting of a threshold to measure how flaky a test case actually is. In particular, the author suggests re-running a failing test multiple times and computing the ratio between successful and failing outcomes: if this ratio is lower than a certain threshold, then the test can be considered flaky and further diagnosed. In other terms, the resource suggests detecting a flaky test by means of its failure rate.

As a conclusion, we might observe that the RERUN approach seems to be the most widely adopted one. Rather than performing it manually, the considered resources let emerge the existence of various tools and frameworks that offer the possibility to automate and configure the re-execution of test cases.

"Fixing Strategy". Only R5, provided insights into addressing flaky tests in mobile environments. The strategies discussed by the author are three: (1) Cleaning up the local environment; (2) Ensuring consistent global state and configurations; and (3) Verifying race conditions. On the one hand, it is worth noting that, from a practical standpoint, these do not really represent proper fixing strategies for test flakiness but development best practices—in other terms, the grey literature review did not let emerge any real fixing strategy to deal with flaky tests in mobile apps. On the other hand, the suggested practices may be used to address some of the root causes that emerged from our analysis. This seems to suggest that flaky tests might be mitigated or even fixed using test code quality strategies: this aspect informed the design of the follow-up survey study, in which we inquired mobile developers on the relation between test code quality aspects and the emergence of test flakiness.

[5]The WALDO tool: https://www.waldo.com/

"Developer's Perspective". Resource R4 concerned about the `droidcon` community.[6] This is one of the largest communities around ANDROID development that periodically organizes conferences around the world. While only one resource highlighted the prominence of flaky tests in mobile apps, the resource should still be considered impactful and representative of a larger number of developers involved in developing mobile apps. In the April 2022 conference held in Lisbon, one of the speakers (Sinan Kozak, an ANDROID engineer in the DELIVERY HERO company) showed some best practices to increase test stability. This talk was required because, according to him, *"the* ANDROID *developers use the flakiness word more than the stability word while talking about* ESPRESSO *tests"*. In other terms, the speaker highlighted that developers performing UI testing often have to do with flakiness, since most tests are born flaky before being made stable by developers. This resource makes us recognize that the problem of flakiness in mobile applications might be managed differently than in traditional systems since it seems to have a different frequency and impact.

Connecting the dots...

The findings coming from the systematic grey literature review allow us to address the research questions posed in our empirical study in different manners.

👉 First and foremost, we observed that flaky tests might be more prominent in the mobile context for two main reasons. On the one hand, the tight relation between an app and its user interface might cause additional issues when it comes to flaky tests, hence confirming the findings by Thorve et al. [110]. On the other hand, the analysis of resource R4 let emerge an additional perspective: especially when performing UI testing, mobile developers might deal with some sort of flakiness already when developing tests in the first place. Hence, in terms of $\mathbf{RQ}_1$, we can report that the systematic grey literature revealed the problem of flaky tests is prominent. We cannot, instead, provide any insights into the problematic nature of flaky tests; this is something that will be explored further through the survey study.

👉 As for the root causes ($\mathbf{RQ}_2$), our study highlighted how some of them are similar to those identified in previous research targeting traditional software development, e.g., race conditions or resource leakage. At the same time, additional problems can be caused by tests verifying user interfaces or making use of third-party libraries. In addition, our grey literature review confirmed the findings by Romano et al. [99]: in some cases, multiple root causes might co-exist, making the flaky test diagnosis process harder for mobile developers.

👉 Finally, with respect to $\mathbf{RQ}_3$, our analysis revealed the existence of some testing tools and/or frameworks that allow developers to verify the presence of flaky tests by means of re-running them multiple times. Perhaps more importantly, practitioners do not discuss about proper fixing strategies but only apply a limited amount of mitigation actions to reduce the effects of flaky tests, e.g., by disabling animations in UIs.

[6] https://www.droidcon.com/

5. Analyzing the Developer's Perception

The second step of our investigation consisted of a survey study, whose details are discussed in the following.

5.1. Survey Design and Structure

We followed the guidelines by Kitchenham and Pleeger [56] to design a survey that balanced the need to be short enough and the requirement of being effective to address the research questions of the empirical study. In the following, we reported the questions included in the survey and the type of answer requested from participants. The various answer options can be found in the online appendix [95].

Survey design. First, we included a brief introductory text where we presented ourselves and the overall goals of the empirical study. In doing so, we defined flakiness as *"a phenomenon occurring when a test case is non-deterministic and exhibits both a passing and failing behavior when run against the same production code"* [72]. This was needed to inform participants of our expectations, the expertise required to participate, and the research objectives. Participants who were not familiar with or not confident enough to discuss test flakiness could decide to abandon the survey already at this point. Furthermore, we provided information on the survey length, which was estimated at about 15/18 minutes and later empirically assessed through a pilot study (more details later in this section). We also explained that the participation was voluntary, that participants could abandon the survey at any time, and that all responses would have been anonymous to preserve the privacy of participants.

We finally concluded this part by asking participants the explicit consent to use the collected data for research purposes and their authorization to proceed with the survey.

Participant's background. The first section of the survey was related to the participant's background. This was required to (1) characterize the sample of developers answering the survey and (2) understand whether and which answers should have been removed because of the poor experience/expertise of some participants. Table 5 reports the list of questions related to this part. In particular, we asked for information on the traditional and mobile programming experience (in terms of years), if participants mainly developed in open-source or other contexts, e.g., industrial, the environment in which they developed mobile applications, i.e., continuous integration and continuous delivery, how much and which testing they typically do when developing mobile applications, which frameworks they use, and whether they have ever dealt with test flakiness in the traditional and mobile development. We designed the latter question to discriminate the participation in the survey. Developers who have never dealt with a flaky test in mobile were deemed unable to proceed - in these cases, we showed a *"Thank You!"* message and let participants leave the survey.

Prominence of test flakiness. The participants who continued the survey were then moved to the second section, which proposed questions concerned with the prominence of flaky tests, and their harmfulness, other than additional contextual information that allowed us to elaborate on the potential co-factors associated with the emergence of test flakiness. As shown in Table 6, we specifically inquired about the frameworks participants were

Table 5: List of questions for the background section in the survey with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 1: Participant's background		**Type**
#1	How long have you been a developer?*	Multiple choice
#2	How long have you been developing mobile apps?*	Multiple choice
#3	What kind of developer are you?*	Checkboxes
#4	You define yourself as ...*	Multiple choice
#5	When you develop mobile apps, do you work in a CI (Continuous Integration) environment, i.e., the practice of merging all developers' work copied to a shared repository several times a day?*	Multiple choice
#6	When you develop mobile apps, do you work in a CD (Continuous Delivery) environment, i.e., an approach in which teams produce software in short cycles and the deployment of a new version is automatic and does not need a human hand?*	Multiple choice
#7	How much testing do you typically perform in relation to production code?*	Multiple choice
#8	How much do you test your app with respect to the following types of testing? To answer this, consider all tests you have done for a mobile application.*	Multiple-choice grid
#9	What framework do you usually use to develop test cases in mobile apps?*	Checkboxes
#10	To what extent have you ever identified flaky tests in the code for **traditional software systems** (non-mobile systems) you were developing?*	5-point Likert scale
#11	Have you ever found yourself identifying flaky tests in the code you were developing, i.e., a test that is non-deterministic and exhibits both a passing and failing behavior?*	Multiple choice

using when a flaky test arose and at which point of the development is test flakiness typically discovered (e.g., during code review or as a result of regression testing).

Table 6: List of questions for prominence and relevance section into the survey with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 2: Prominence and Relevance of Flakiness		**Type**
#12	How often have you found flaky tests in your projects (i.e., from less than a few times per year to daily)?*	5-point Likert Scale
#13	How dangerous do you consider the flaky tests identified in terms of the failures they have caused?*	5-point Likert Scale
#14	What framework were you using when you detected the flakiness?*	Checkboxes
#15	Which point in the development pipeline did you identify flakiness in your project?*	Checkboxes

Root causes of test flakiness. Afterward, the survey proposed questions related to the root causes of test flakiness—Table 7 shows the list of questions related to this section. These questions aimed to stimulate participants to think about the characteristics of the flaky tests they dealt with. We first asked developers to indicate the most common root causes from their perspective, along with the complexity in terms of detection and fixing. Then, we asked the frequency with which they identified a root cause in both traditional and mobile development, giving the opportunity to answer *"I do not remember"*. When answering these questions, they could select one or more options from a predefined list of the most common root causes coming from both previous works

[26, 61, 62, 72, 110, 125] and the results of the systematic grey literature review previously conducted. Developers could also indicate additional root causes not included in the list through the *"Other"* option.

In addition, we asked whether the participants identified some relations between the emergence of a flaky test and general properties of the test code, like a high number of lines of code, the use of external resources, and so on. The proposed list of properties comes from the results of a recent work by Pontillo et al. [94]: the authors assessed the relationship between flaky tests and a set of test and production code metrics and smells, reporting that some of these factors might impact the likelihood of a test being flaky. By presenting these properties in the survey, we could challenge the findings by Pontillo et al. [94], complementing them with information on the developer's perception. Also, we could further verify the insights coming from the systematic grey literature review, where we discovered that some properties of production code might affect test flakiness. We did not explicitly mention the name of the metrics and smells, but reported their description: this was done to mitigate risks due to confirmation bias [57], e.g., developers might have been more inclined to identify a relationship if they were aware of the fact that some of these properties referred to sub-optimal test code design and/or implementation choices. Participants had also the chance to write other answers to report on relations they identified in their past experiences.

Table 7: List of questions concerning the root causes of flaky test in the survey with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 3: Root Causes of Flaky Tests		**Type**
#16	Could you indicate which root causes you have most identified and analyzed in the mobile application?*	Checkboxes
#17	How many times did you identify/analyze the following root causes after detecting a flaky test during the development of a **traditional software system** (non-mobile application)?*	Multiple-choice grid
#18	How many times did you identify/analyze the following root causes after detecting a flaky test during the development of a **mobile application**?*	Multiple-choice grid
#19	Based on your experience, which flaky tests are more difficult to DETECT in terms of effort?*	Multiple-choice grid
#20	Based on your experience, which flaky tests are more difficult to FIX in terms of effort?*	Multiple-choice grid
#21	When you detected test flakiness in your mobile app, did you ever recognize the following characteristics?*	Multiple-choice grid
#22	Did you detect any additional characteristics other than those previously presented?*	Paragraph

Diagnosing of test flakiness. The next section of the survey was related to the strategies employed once a seemingly flaky test was identified, e.g., when a test that previously succeeded failed—Table 8 reports the list of questions. We first asked how participants verified the actual flakiness of a test (e.g., by rerunning it multiple times), how the development pipeline moved forward (e.g., by temporarily disabling the test), whether the same flaky test appeared again in later stages of the development and, if so, how participants reacted. When answering these questions, the developers could check one or more of the predefined answers that we included to ease the survey compilation. The predefined answers were based on the strategies described by previous works [26, 64, 65] and those coming from our systematic grey literature review. At the same time, developers could still provide additional answers through the *"Other"* option.

Table 8: List of questions entered in the survey for the flakiness detection strategies in the mobile apps, with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 4: Detection of Flakiness in Mobile Apps		**Type**
#23	Once you identified a failure how did you later diagnose that it was a flaky test?*	Checkboxes
#24	Once a flaky test is identified, how does the development pipeline move forward?*	Checkboxes
#25	After you have found and fixed one or more flaky tests, to what extent have you found again the same flaky test going on with the development? Please address the question independently from the difficulty of fixing the flaky test.*	5-point Likert Scale
#26	If you answered with at least 3 to the previous question, could you describe what you did in the development process (e.g., you fixed the flaky test again, you disabled the flaky test, you rerun the flaky test)?	Paragraph
#27	Would you like to give us further information about the flakiness detection process you used?	Paragraph

Fixing of test flakiness. After the diagnosis, we focused on the fixing process. Table 9 reports the list of questions. Similar to what was done earlier, we asked how participants typically fix flaky tests. Then, we provided them with a list of fixing strategies for each of the root causes presented in the previous section of the survey, asking participants to indicate how often they applied those fixing strategies to address flaky tests. We exploited the findings by Eck et al. [26] to compile the list of fixing strategies: they have indeed provided a taxonomy of the most frequent operations performed by developers to deal with test flakiness. We did not use the results of the systematic grey literature review when proposing potential fixing strategies, as it did not let emerge any proper strategy to consider—as discussed in Section 4.4. Our participant also had the chance to answer *"I do not remember"* and indicate additional strategies to put in place. Finally, we asked the participants to give us more details about the fixing process in a *Continuous Integration* and *Continuous Delivery* environment, asking them if there were any similarities and differences based on their experiences.

Survey closure. In the last section of the survey, before thanking the participants, we allowed them to enter their e-mail addresses to (1) receive a summary of our results and/or (2) participate in a future follow-up semi-structured interview. As shown in Table 10, participants were also allowed to enter one or more links to open-source repositories of the apps they developed: this was part of the incentives we provided for the participation, namely a free test code quality analysis of the repositories shared. It is worth clarifying that the participants were made aware of the offer only upon completion of the survey. In other terms, we did not advertise the test code quality consultation in the call for participation but rather we presented the offer just as a recompensation for the effort and time the participants spent in answering our questions.

At the end of each section, we included a free-text answer where developers could elaborate and provide additional information on how they deal with flaky tests.

Survey validation. A crucial aspect to consider when designing surveys is concerned with their quality and the time required to fill them out. While longer surveys provide more insights into the matter, the excessive length

Table 9: List of questions related to the fixing strategies employed by mobile developers in the survey with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 5: Fixing Flakiness in Mobile Apps		**Type**
#28	How often have you fixed flaky tests in your projects (i.e., from never to daily)?*	5-point Likert Scale
#29	How often have you used these strategies to fix a flaky test?	Multiple-choice grid
#30	Did you use any additional strategies other than those previously presented?*	Paragraph
#31	If you work in a CI environment (i.e., the practice of merging all developers' working copies to a shared repository several times a day), once a flaky test is detected and fixed, do you immediately release a new version of the application?*	Paragraph
#32	If you work in a CD environment (i.e., an approach in which teams produce software in short cycles and the deployment of a new version is automatic and does not need a human hand), how do you perceive the presence of a flaky test? Do you quickly try to fix and update the mobile app, or knowing that the test is occasionally successful make you proceed differently?*	Paragraph
#33	In your opinion, what do you think are the similarities and/or differences in the flakiness management process (i.e., detection, mitigation, and fix) in mobile applications between CI and CD environment?*	Paragraph
#34	Would you like to give us further information about the fixing process you used?*	Paragraph

Table 10: List of questions related to the fixing strategies employed by mobile developers in the survey with the type of response provided. The asterisk next to some questions indicates that it was mandatory to enter an answer.

Section 6: Survey Closure		**Type**
#35	Would you like to learn about the results of our study or to be contacted to participate in a follow-up interview/focus group on this topic? If yes, please write down your email.	Short Answer
#36	If you are an open-source developer, would you like to share the mobile app code where you spotted one or more flaky tests? If yes, please write down the link to the repository.	Paragraph
#37	Would you like to share a link to other open-source projects you're working on that you haven't experienced any flakiness issues with so far? If you're interested, we can review the code and get back to you if we spot any flaky tests. Please, write down the link to the repository.	Paragraph

may discourage participation [5]. We took great care of this point and, for this reason, after defining the first version of the survey, we conducted a pilot study [78]. A pilot study typically consists of an experiment with a small sample of trusted participants who might provide feedback on the length, clarity, and structure of the survey. In our case, the pilot study was conducted with four mobile software engineering developers with experience in software development and testing, who were selected from our contact network. The four involved developers well matched the ideal population of our survey, i.e., they have expertise in software testing and deal with test flakiness in the context of their projects.

They were contacted via e-mail by the first author of the paper and, upon acceptance, were instructed on how to fill out the survey. Besides the link to the survey, the four mobile developers were provided with a text document where they could report their observations in terms of length, clarity, and structure of the survey. We also

requested them to clock their progress so that we could have precise indications of the time required to complete the survey. They had one week to complete the task. The four pilots sent their notes back to the first author upon completion. These were later jointly analyzed by the first two authors of the paper, who identified and addressed a few clarity issues in the description of the various section of the survey. Also, one of the pilots let emerged a lack of emphasis on the development that needed to be relied on to answer the question, i.e., traditional software system or mobile application development. We recognized this problem and emphasized the concept in questions #17 and #18. In terms of timing, the pilots successfully completed the survey within 17 minutes. We finally double-checked our changes with the pilots, who confirmed that all their feedback was addressed.

5.2. Survey Recruitment and Dissemination

The following sections elaborate on the decisions taken to maximize the developer's participation.

Survey Recruitment. A key choice of survey studies is concerned with the selection of suitable target participants [5]. We consciously avoided spreading our survey on social networks, as we could not have had control over the answers received [5]. On the contrary, we identified PROLIFIC[7] as a useful instrument to recruit our participants. This research-oriented web-based platform enables researchers to find participants for survey studies. It allows the specification of constraints over participants, which enabled us to limit mobile developers' participation. PROLIFIC implements an *opt-in* strategy [50], meaning that participants get voluntarily involved. To mitigate the possible self-selection or voluntary response bias, we introduced a monetary incentive of 7 USD. Incentives are well-known to mitigate self-selection or voluntary response bias, other than increasing the response rate, as shown in previous studies targeting the methods to increase response rate in survey studies [48, 100].

The PROLIFIC platform has been studied by various researchers [25, 98]. For example, Reid et al. [98] recently defined recommendations to conduct surveys using the platform. We followed those recommendations while preparing the survey, e.g., we pre-screened participants in order to assess their suitability for the study.

Survey Dissemination. The recruitment platform allows to include external links to web pages hosting the actual survey to fill. We implemented our survey through a GOOGLE form[8] and included it within PROLIFIC. We collected 153 responses, which were subject to quality assessment—more details are discussed in Section 5.3.

Ethical Considerations. In our country, it is not yet mandatory to seek approval from an Ethical Review Board when releasing surveys with human subjects. Nevertheless, when designing the survey we mitigated many of possible ethical and privacy concerns [46]. We guaranteed the participants' privacy by gathering anonymous answers. Participants voluntarily provided their e-mail addresses to receive a summary of the results and/or request the free test code quality analysis on their apps. When recruiting developers, we stated the goal of the survey study, other than explicitly reporting that the given answers would have been used in the scope of a research activity that would

[7] PROLIFIC website: https://www.prolific.com/.
[8] GOOGLE form website: https://www.google.com/forms/about.

not have any intention of publishing sensitive data. Finally, we clarified that completed surveys would eventually become public, preserving the privacy of the participants.

5.3. Data Cleaning and Analysis

Once we had collected the responses, we performed a quality assessment phase. In particular, the first author of the paper went through the individual responses collected to validate them, possibly spotting cases where the participants did not take the task seriously or did not have enough experience to provide valuable insights. This procedure led to the exclusion of 23 responses, that are reported in the online appendix [95]. On the other 130 answers, we first make sense of the data by analyzing the closed answers. As the reader might see by looking at the detailed set of questions available in Section 5.1, most questions were formulated so that participants could express their opinions through check-boxes or using a 5-point Likert scale [82] with different ranges, e.g., from *Very rarely* to *Very frequently*. These answers could be analyzed using statistical analysis, collecting numeric distributions and interpreting them via bar plots and tables.

Instead, the open answers were subject to content analysis [18], a research method where one or more inspectors go over the data of interest and attempt to deduct their meaning and/or the concepts they let emerge. The process was conducted by the first two authors of the paper, who jointly analyzed the individual responses to identify and label the main insights and comments left by participants. The content analysis process took around 80 person/hour and was used to better understand and contextualize the position of participants with respect to the diagnosing and fixing processes adopted when dealing with flaky tests.

5.4. Analysis of the Results

This section summarizes the demographics of the 130 involved participants, other than addressing the three research questions of the study.

Demographics. Figure 3 overviews the main information on the participants' backgrounds. In the first place, a large portion of participants qualify themselves as freelance or open-source developers (36.6% and 28.2%, respectively), while 35.1% and 26.1% of them consider themselves as industrial and startup developers, respectively. Of the 130 responses, 32 participants qualified themselves as belonging to more than one category: 11 participants recognized themselves as open-source and freelance developers, 7 as open-source and startup developers, 4 as open-source and industrial developers, 4 as startup and freelance developers, 2 as industrial and startup developers, 2 as open source, startup, and freelance developers, 1 as industrial and freelance developer, and finally 1 as industrial, freelance, and startup developer. As a consequence, the percentages reported in the paper do not sum up to 100%. Almost 59% of the participants have more than 3 years of development experience and 24% of them have been developing mobile applications for more than 3 years. 79% of the participants were ANDROID developers, with others involved in developing apps using IOS, FLUTTER, or other. Concerning the environments in which the participants work, 71% declare that they develop in a Continuous Integration environment, while 50% work

in a Continuous Delivery environment—40% of them develop in both Continuous Integration and Continuous Delivery. More important for the objectives of our study is, however, the information provided on the amount of testing that participants typically perform when developing their apps. Concerning the answers to question #4, we can observe that 69 participants (66.9%) reported doing testing for less than 60% of their production code. Finally, participants declared to develop between 21% and 40% of test cases for all types of testing levels.

Based on these pieces of information, we can conclude that our sample is quite heterogeneous and composed of mobile developers working in different contexts and having various levels of seniority. On the one hand, these demographics are in line with the recent findings reported by Pecorelli et al. [90], who showed that mobile apps are typically poorly tested in practice—this consideration will be important to interpret some of the results discussed later in this section. On the other hand, it is worth remarking that Pecorelli et al. [90] only dealt with open-source ANDROID applications, while our work also collects information on mobile apps developed in industrial and startup environments, other than in different operating systems. For this reason, the results we will report in the next sections include the perspective of a larger population of mobile developers.

Despite the relatively low amount of testing done, 64.6% of developers have experienced test code flakiness at least once in the projects they developed. This percentage already suggests the diffuseness of the problem, indicating that test flakiness is not a negligible issue. More particularly, 84 participants claimed to have dealt with flaky tests and were allowed to answer the subsequent questions of the survey, which aimed to understand better the flakiness problem in mobile development. The remaining 46 participants were thanked for their participation and left the survey, i.e., we discarded those developers from the subsequent analyses that led us to address the research questions. Among these 46 participants who were not considered, 7 (15%) of them also declared that they have faced flaky tests in traditional contexts often or very often, 8 (17%) sometimes (46%), 21 rarely, and only 10 (22%) never. We found these data interesting, as they seem to emphasize the existence of significant differences between the problem of flakiness in mobile and non-mobile applications - hence motivating the need for mobile-specific investigations like ours.

RQ_1 - On the Relevance of the Problem. Figure 4 shows the answers provided by participants to questions #12 and #13, which allowed us to address **RQ_1**.

Regarding the frequency of flaky test appearance, 43% of the participants indicated that flakiness *sometimes* arises, i.e., on a monthly basis, while 12% claimed to find flakiness *frequently*, i.e., weekly and more. The last results for question #12 show that 33.3% of participants claimed that flaky tests are *rarely* observed, i.e., a few times per year, while the 12% declared that flaky tests are *very rarely* observed, i.e., less than a few times per year.

In the first place, these results suggest that, despite the low amount of testing performed, the flakiness issue is observed in practice, hence corroborating the insights coming from our systematic grey literature review. Moreover, our findings are in line with what has been investigated in traditional software: for instance, Eck et al. [26] reported that over 40% of developers faced the problem only a few times per year.

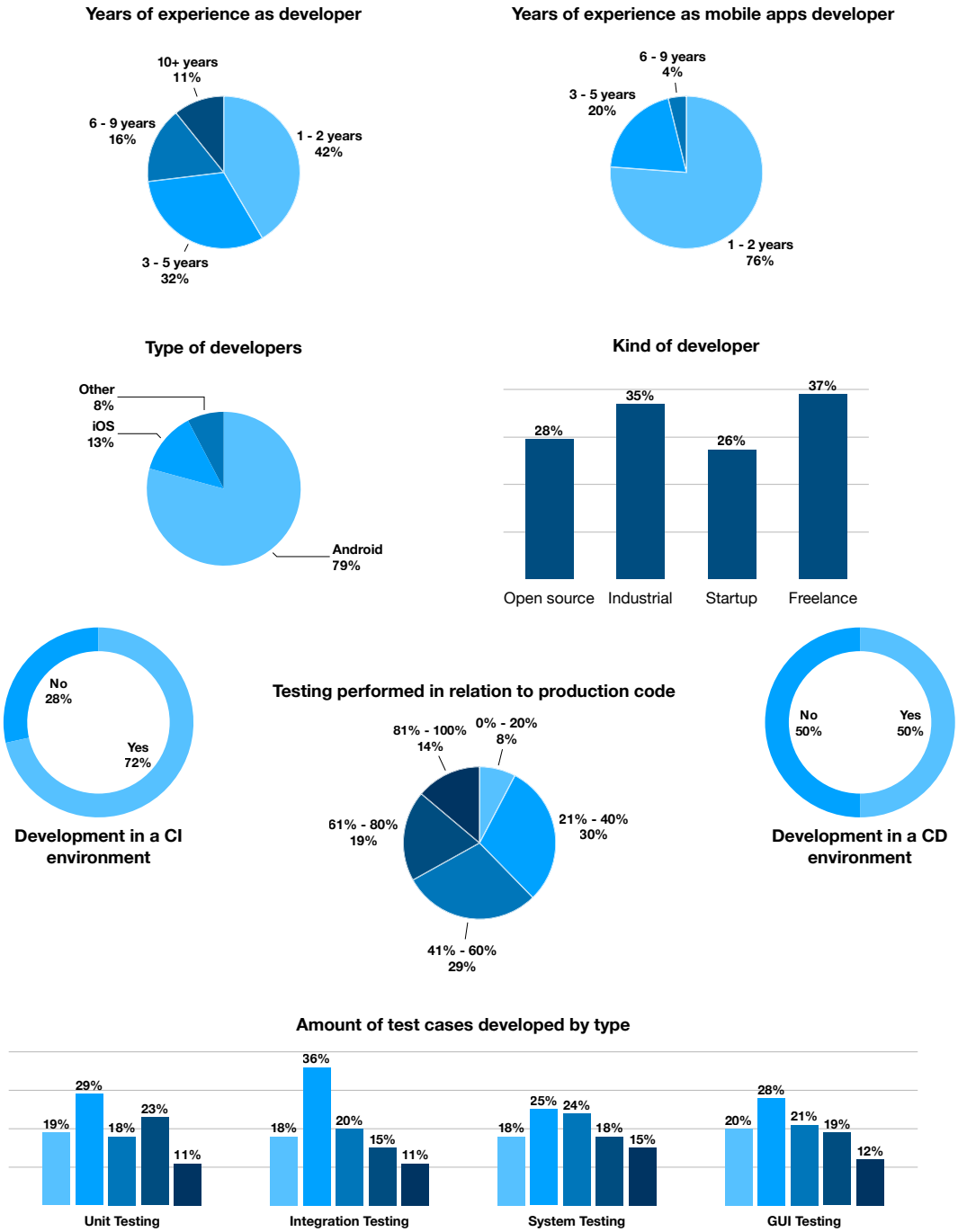

Figure 3: Graphics of the background of our participants.

The answers received on the harmfulness of flaky tests in terms of the failures they caused (question #13) corroborate the idea that the flaky test problem is serious. Around 45% of the participants reported that flakiness is *dangerous* or *very dangerous*, while 52.4% indicated that the problem could cause *moderate* issues. Hence, only

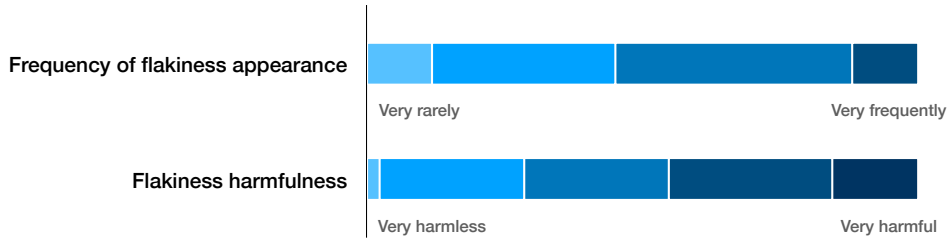

Figure 4: Results for questions #12 and #13, i.e., how often they have detected flakiness in their project and how dangerous they consider the failure caused by the flaky test to be.

a limited number of developers (2.4%) considered the problem a minor issue.

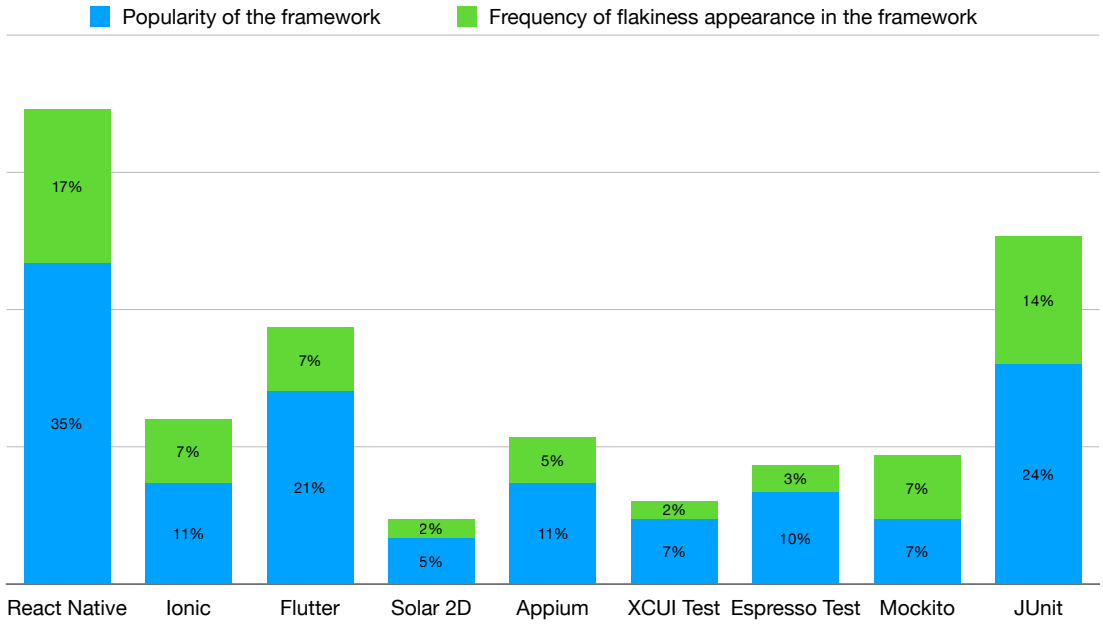

Figure 5: List of frameworks where flakiness appears during development—some are for iOS development (XCUI Test) or Android development (Solar 2D, Mockito and JUnit), others are for cross-platform development (Appium, React Native, Flutter, and Ionic). The blue bar indicates the popularity of the framework in the participants' responses, while the green bar shows the frequency of flakiness appearance in a specific framework based on the responses.

As for the testing frameworks used when flaky tests were discovered (question #14), Figure 5 shows the results obtained. For Android development, participants mentioned frameworks such as Appium, Solar 2D, and Espresso Test, while for the iOS development, we obtained as responses only XCUI Test. In addition, frameworks like Flutter and React Native—which can be used for creating cross-platform apps—were also mentioned. Analyzing the frequency of appearance of flakiness in the various frameworks based on their popularity as reported by the participants, we could see that there seems to be a correlation—the more popular a framework

is, the higher the risk of flakiness. Unfortunately, the responses obtained and analyzed are too limited to allow for more detailed statistical analysis, but we plan to investigate this in our future work.

Finally, Figure 6 shows the results when analyzing the context where flakiness is detected (question #15). In most cases, developers discover flaky tests from bug reports (54%), through code review activities (48%), or regression testing (40%). Most likely, these results reflect the typical processes applied to guarantee software quality assurance: indeed, developers use to automate the opening of bug reports when previously successful test cases suddenly fail [119], when tests are run against production code in a continuous integration setting [27, 114], or as a consequence of the code review activities performed on production code, i.e., when executing tests to verify source code during code review [106].

Figure 6: Answer to question #15, i.e., at which point of the development pipeline the participants identified the flakiness? Please, consider that the sum of the percentages does not make 100 because participants could answer with more options to the question.

RQ$_2$ - On the root causes of test flakiness. Figure 7 overviews the most common root causes of flakiness along with their perceived detection and fixing effort. As shown, the most common pertains to *API Issues* and *Program Logic*. On the one hand, these results contrast what has been reported by previous works [26, 72]. Both indeed relate to issues in the *production code* rather than in the test code, which implies that the sources of flakiness are often to be searched in how the app logic is implemented or how external services are integrated. On the other hand, these findings corroborate what we discovered with the systematic grey literature review, i.e., production code factors and third-party libraries might impact the emergence of flaky tests.

The difference with respect to the knowledge acquired in traditional software might be a reflection of the peculiarities of mobile development. The continuous release approach followed by developers to introduce new features and fix defects encountered by users [81] might enforce them to induce sources of non-determinism in the production code more frequently. Previous works [22, 69] have indeed shown that the continuous changes applied to mobile applications are correlated with an increase in maintainability, reliability, and security concerns, which are all well-known causes of instability that may complicate software testing activities [30]. In other terms, our findings suggest that test cases mostly become or might become flaky as a consequence of poor design solutions applied by developers when evolving mobile apps. The API issues mentioned by our participants may provide additional motivations for the conclusions drawn so far. In fact, mobile apps make intensive use of REST

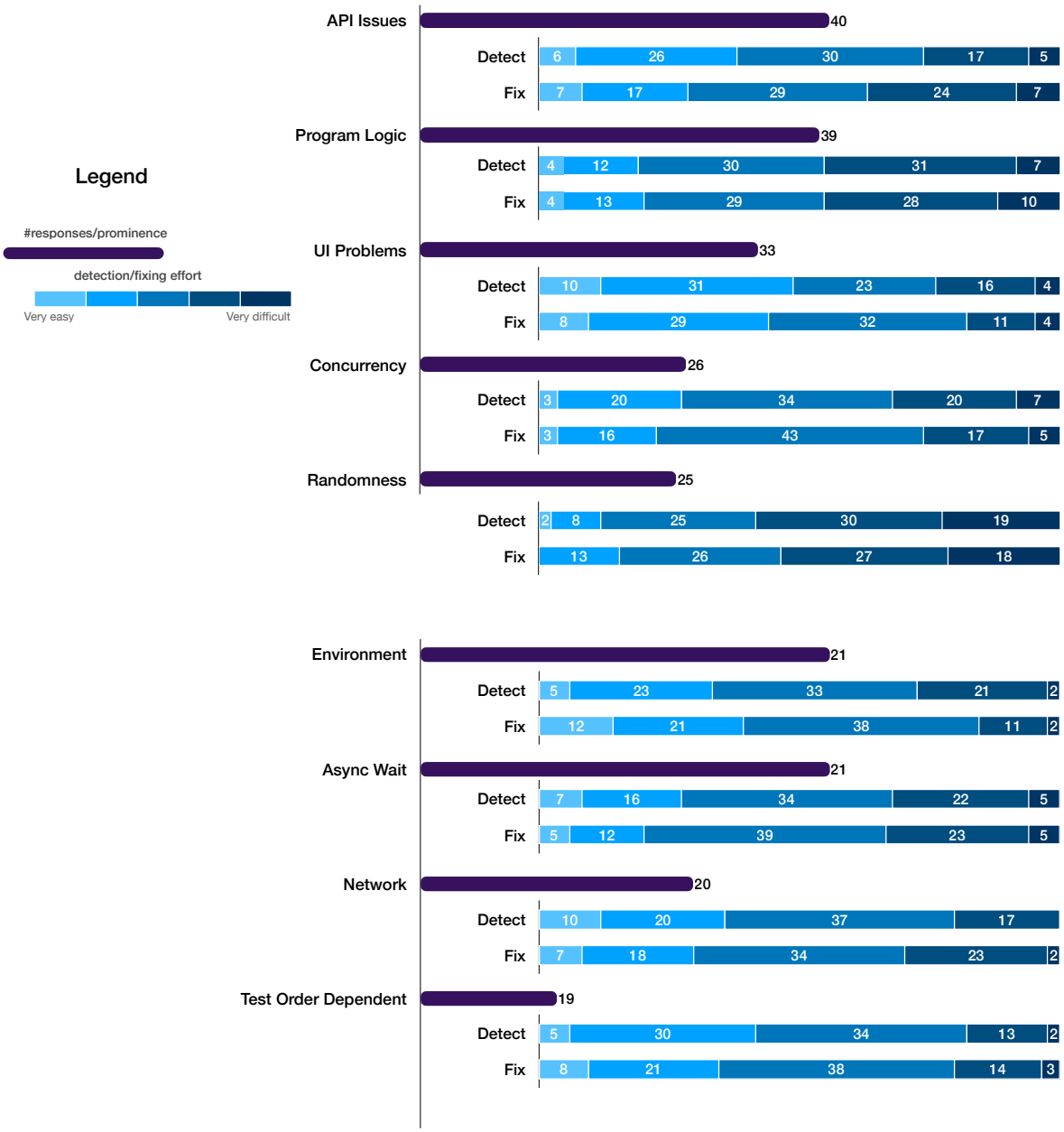

Figure 7: The most common root causes identified by participants. For each root cause, we reported the developer's perception in terms of detection and fixing effort, i.e., questions #16, #18, and #19.

APIs [83], which are often hard to use [1, 3] or even change- and fault-prone [9, 101]. This aspect may create additional issues and indirectly affect the quality and effectiveness of test cases. Interestingly, the two root causes

discussed are those that developers perceive as the hardest to detect and fix. This result suggests the need for further experiments that focus on the source code quality perspective of test flakiness.

The third root cause most identified in our study is *UI problems*—this result confirms the results of our grey literature review and the results showed by Thorve et al. [110]. Other answers provided by developers revealed root causes similar to those identified in previous work conducted in the context of traditional systems [72]—this confirms, once again, the findings coming from the systematic grey literature review. Problems connected to randomness, network, test environment, concurrency, and test order dependency are well-known in the research community—and various automated techniques to deal with them have also been proposed [23, 34]. Nonetheless, we could observe differences in terms of frequency of appearance. For instance, with respect to Eck et al. [26] and Luo et al. [72], problems due to test order dependency occur way less, according to the opinions of our participants. This difference is also visible when considering the effort required to detect and fix them, which is *medium* for most participants. These results further suggest the need for specific, contextual empirical investigations aiming to understand mobile app testing characteristics. The involved developers did not mention root causes different from those proposed in the survey, meaning that, in their perspective, the list was complete.

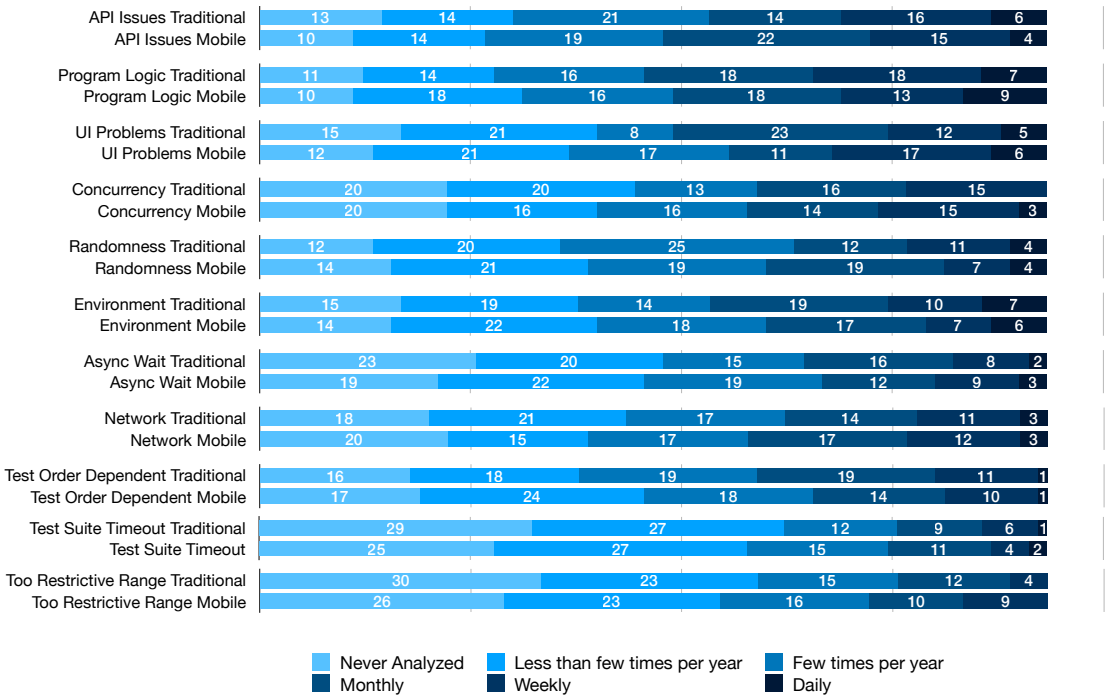

Figure 8: Comparison between the root causes analyzed by developers in the traditional software system and in the mobile applications, i.e., questions #17 and #18.

Figure 8 shows the comparison between the most common root causes analyzed in the traditional software system and the mobile applications. We can observe that the flakiness related to *API Issues*, *Program Logic*, and *UI*

problems are more diffused in mobile applications than traditional systems, respectively with 49%, 48%, and 40% of the responses varying between *Monthly* and *Daily*. Analyzing the other root causes presented in the survey for questions #17 and #18, we can see that the frequencies with which they occur in traditional systems and mobile applications are similar. We can therefore conclude that some root causes are more likely or more frequent to arise in mobile apps, justifying the need for further investigations into the matter.

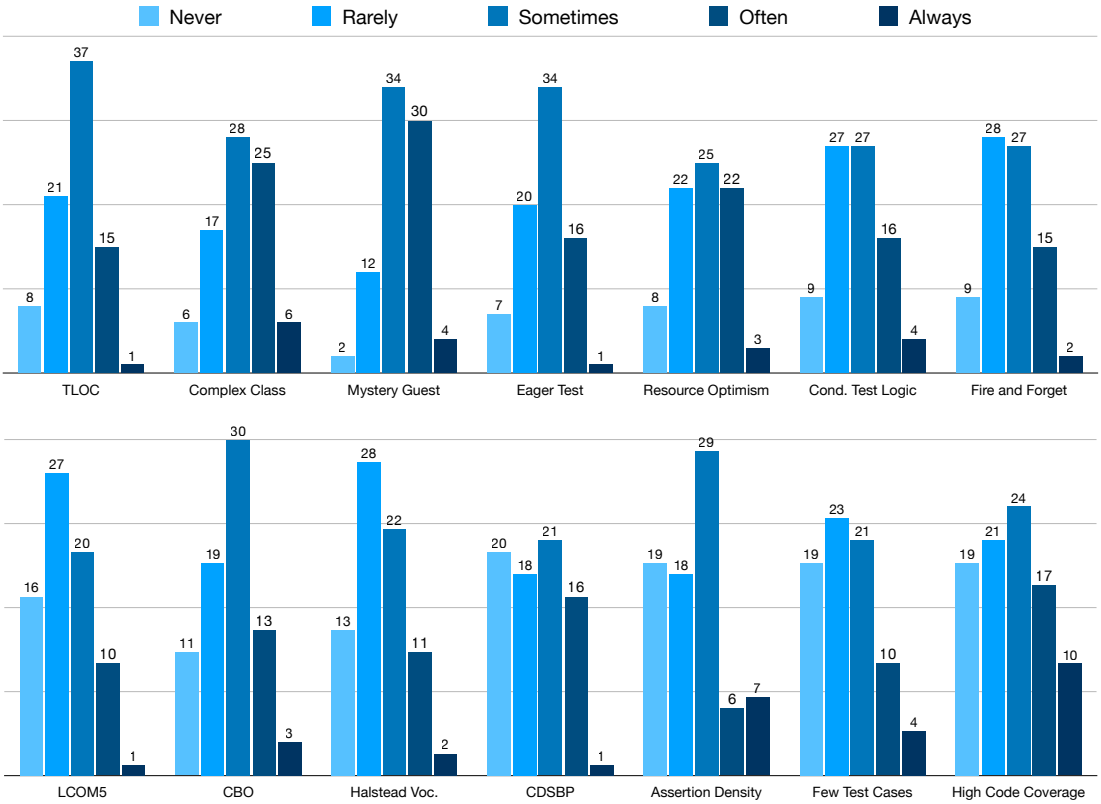

Figure 9: The frequency of appearance of the test and production code factors provided in question #21.

Perhaps more interesting were the answers provided when linking flaky tests to test code metrics and smells (question #22) - Figure 9 shows the results achieved. From this analysis, we could recognize that all features proposed in our question are to some extent, related to the presence of test flakiness. For instance, design concerns pertaining to the improper management of external resources, i.e., *Mystery Guest* and *Resource Optimism*, appear in the 81% and 60% of the cases with values between *Sometimes* and *Always*. Other factors that seem to be related to test flakiness are *Eager Test* (61% of the responses between *Sometimes* and *Always*) and the number of lines of code in the test suite (63% of the responses between *Sometimes* and *Always*). These are well-known problems in the field of test code quality [55, 107, 111], other than being already associated with test flakiness [17, 94]. On a similar note, other design metrics, like coupling between tests or assertion density, seem to relate to flakiness, pointing out that how test code is designed might affect the likelihood of tests being flaky. Finally, we could also

observe that factors like code coverage or the size of test suites are not frequently connected to flakiness: this seems to indicate that these factors are less relevant and insightful in identifying the sources of flakiness.

RQ_3 - On the diagnosis and fixing strategies. Our third research question targeted mobile developers' test flakiness diagnosis and fixing processes. For the sake of understandability, we split the following discussion based on the lifecycle of a flaky test, i.e., verification, detection, and fixing of flakiness. Besides providing a more structured analysis of the developers' answers, this also enables an easier comparison with existing literature targeting the developer's perception [26] and the lifecycle of test flakiness [62].

Flakiness verification. The responses provided by developers let us first understand how they diagnose the existence of a flaky test, i.e., how they verify that a failing test is intermittent. In particular, the participants claimed to use two mechanisms, namely the so-called RERUN [64] and the manual test code inspection [106], respectively in the 81% and 74% of the cases. The former result is in line with the grey literature analysis and consists of rerunning tests multiple times with the aim of assessing their degree of reliability. One documented issue about this strategy concerns the lack of insights on the number of times a test should be rerun to verify its flakiness [24, 75]. This is also perceived by the mobile developers surveyed: for example, developers #37 and #78 reported that they *"Rerun the test multiple times under different conditions and using different values"*.

The manual test code inspection relates to the manual debugging of the potential issue affecting the test code, with the aim of comprehending the nature of the failure. In this respect, only 23.8% of the participants declared to use dedicated tools (question #23). This result could show that developers do not know about the existence of some tools or frameworks that can be helpful in the analysis of flaky tests. For example, developer #38 report that they *"do not know if there is any targeted tools for flakiness detection.[...]"*.

Flakiness detection. Once developers verify that a test is flaky, they tend not to ignore it. Only 14.3% of the participants declared that they voluntarily disabled one or more flaky tests in the past, while 13% ignored the sources of flakiness and proceeded with the development without caring about the issue. The vast majority of the surveyed developers (89%) reported their attempts to fix a flaky test as soon as it arises. This process is typically approached manually because of the lack of tools that might assist them in this process—our participants could not provide us with any sort of automation they were aware of.

More worrisome is what emerged from the answers to question #25. Indeed, 48.8% of participants periodically identified the same flaky tests over the evolution of their app. This means that a non-negligible amount of flaky tests, which were supposed to be fixed, arise again in a later stage of software development. On the one hand, this result suggests that developers cannot always properly verify the outcome of a fix, perhaps because of the lack of usable tools [86]. On the other hand, this aspect might be connected to what was discovered in the context of RQ_2: if most flaky tests are due to problems pertaining to production code, it seems clear that developers trying to fix them by modifying the test code are likely to fail and see the flakiness appears again. Our findings point to the well-known problem of locating the source code flakiness to reduce maintenance efforts [86].

The problems due to the recurrence of flaky tests do not limit themselves to the short term. The participants reported a constant interest in understanding and fixing the source of flakiness. 43% of them claimed that they try to fix again the flaky tests and a constant follow-up monitoring of the flaky tests. This is typically realized through the RERUN approach, which is used to control the severity of the problem, namely the extent to which the flakiness keeps manifesting itself. Finally, only two participants claimed that the recurrent flaky tests were disabled or even removed from the test suite.

Flakiness fixing. Turning our focus on the fixing strategies, 42.9% of developers indicated that they *sometimes* fix flakiness, while 8.3% fix flakiness *frequently*.

Concerning the strategies applied by participants to fix flaky tests, answers to question #29 showed that the most frequent strategies used to fix flakiness are *Checking instance variables before that the test is accessed and executed* (61%) and *Verify the race condition* (53.5%). The first strategy is often associated with root causes like *Program Logic*, *Concurrency*, and *Randomness*. The fixing consists of verifying and adapting the status of the instance variables set in the fixture of a test suite so that the related test cases can work in a proper environment. Checking the context of a test case seems to be a pervasive problem in mobile applications, as Thorve et al. [110] also reported. Very likely, this is a reflection of the peculiarities of mobile applications, which rely on external sensors, APIs, etc., being, therefore, more prone to non-deterministic variations of their environment.

For the second most common strategy, the race conditions refer to multiple threads that access the same data without controlling which one is first so that each run can differ. As reported in our grey literature review, verifying the race conditions using the THREAD SANITIZER is a fixing strategy for flakiness depending on *Concurrency*.

Other common fixing strategies concern the removal of concurrency-related problems. The most common in this respect are the *Add waitFor statement* (36%) and the *Add new code, so the conflicting object is destroyed before continuing the execution* (34.5%). Finally, we also noticed that 24% of the participants mentioned that so-called *Skip Non-Initialized Part* strategy: this refers to the addition of code aimed at skipping non-initialized parts and making the test run faster [72]—this represents an alternative strategy to deal with API issues. The additional fixing strategies provided in the survey appear to be only rarely used in practice.

Overall, we could conclude that the analysis of the detection and fixing strategies is in line with the root causes pointed out by the participants. Most of the strategies are connected to very specific problems of mobile applications. Some participants also provided additional insights into the operations done when fixing flaky in a *Continuous Integration* Environment and in a *Continuous Delivery* Environment. First and foremost, the answers reported that independently from the environment the participants try to fix flakiness quickly and release a new version. Only in some cases, participants tend to ignore the flaky tests. For instance, developer #83 reported that *"if the new version is needed for a demo, probably ignore/disable the test, [...] but in general we try to fix/handle the test failure"*. Finally, participants declared that they do not see any differences in the management of flaky tests between the two environments, so we could conclude that test flakiness is a relevant problem regardless of the

process and development environment.

Connecting the dots...

The findings obtained from the survey study allowed us to further elaborate on the research questions.

👉 As for **RQ**$_1$, test flakiness is a frequent problem for 55% of the participants and may lead to harmful consequences. Developers identify flaky tests during bug report (54% of the responses) and code review (48% of the responses) activities. Finally, it seems that the choice of testing frameworks might not impact the emergence of flaky tests. Our results corroborate the systematic grey literature review with respect to the prominence of the problem: in this respect, while the grey literature review could only anecdotally suggest that the problem of test flakiness is relevant for mobile developers due to the limited amount of resources available, the survey study could provide tangible insights in this respect, quantifying the relevance of the problem more precisely. Furthermore, the survey study could provide a larger overview of the problematic nature of flaky tests, which could not be possible to gather solely by looking at the resources retrieved through the grey literature review.

👉 In terms of root causes (**RQ**$_2$), about the 50% of developers indicate issues in production code as the main causes of test flakiness, which is in line with the insights arising in our systematic grey literature review. This is likely due to continuous changes applied or the hardness to deal with external APIs. Also in this case, the survey study had the function of quantifying the insights arising from the grey literature review and suggested that further studies investigating the peculiarities of mobile testing are needed. Finally, the survey results could also extend the results of the grey literature review, letting emerge the role played by test smells as a potential indicator of flakiness.

👉 As for the diagnosis and fixing procedures (**RQ**$_3$), developers typically diagnose test case failures by rerunning test cases multiple times or manually inspecting test code (80%). More importantly, the detection process is mostly manual and conducted with low or no tool support, even though some automation mechanisms emerged from the systematic grey literature review. While the grey literature review let arise the presence of various tools and frameworks to deal with test flakiness, the survey results revealed that these instruments are not actually used in practice: in this sense, the survey study had the role of bridging the gap between the availability of detection and diagnosis instruments and their adoption in practice. Furthermore, the survey study identified additional insights into the state of the practice. First, the recurring emergence of the same flaky tests makes developers less and less inclined to address the causes of flakiness. Second, the key fixing strategies are connected to the verification/update of the instance variables when the test case is executed (61%) and the verification of the race conditions (53.5%)—these strategies did not emerge in the grey literature survey. Both strategies might be somehow considered as best practices when developing and verifying test cases: we might therefore claim that future investigation into the best practices to write and verify test cases might help developers avoid the introduction or speed up the fixing of flaky tests. The survey results in

this respect complemented those of the grey literature review: these may indeed provide an initial catalog of practices applied by practitioners to deal with flaky tests; on the contrary, the grey literature study could not let emerge any specific practice.

6. Discussion and Implications

The results of our empirical study pointed out a number of valuable discussion points that are worth to be further elaborated, along with a number of implications for both researchers and practitioners.

6.1. Contextualizing Our Findings

As discussed in Section 2, the scientific literature on test flakiness is rapidly growing and multiple papers have already assessed the developer's perception of flaky tests, other than proposing tools and techniques to manage them. It is, therefore, worth contextualizing our findings with respect to previous research, to provide the reader with a more comprehensive view of our results and how they impact the current knowledge on test flakiness.

With respect to the root causes of flakiness, our findings extend those reported by Thorve et al. [110]. In particular, according to the surveyed participants, we identified *"API issues"* as the most prominent cause. This was never encountered by Thorve et al. [110] during their manual analysis of 77 commits pertaining to 29 open-source ANDROID apps. On the one hand, the difference might be due to the different research methods employed, which led us to gather the experience of a large amount of mobile practitioners. On the other hand, our sample is not only composed of open-source developers engaging with the ANDROID operating system but also of practitioners working in different contexts and on different platforms. As such, we cover a larger amount of environments, hence extracting additional knowledge with respect to the state of the art. At the same time, it is also worth remarking that other prominent causes of flakiness identified by Thorve et al. [110] were also frequently mentioned by the practitioners involved in our study - this is the case of *"Program Logic"* and *"UI problems"*. In this respect, our findings corroborate those of Thorve et al. [110] and show that these causes are actually relevant in practice. A similar discussion can be drawn when considering the least frequent root causes. For instance, flakiness due to *"Test Order Dependency"* was confirmed to rarely arise in mobile applications.

When analyzing the resources from the systematic grey literature review, we found that in some cases the flaky tests due to *"UI problems"* were instead a consequence of other root causes. In other terms, what developers believe is due to issues with the design and rendering of widget layouts hides more sneaky issues. While this was already observed by Romano et al. [99], there are two considerations to make here. On the one hand, our findings triangulate those obtained by Romano et al. [99] using a different research approach (survey study rather than a manual analysis of UI tests): in this respect, we could confirm that test code flakiness is particularly relevant when considering UI tests. On the other hand, however, we complemented those findings by reporting that, in most cases, practitioners could not recognize the specific root cause making a UI test flaky: in this sense, our

work emphasized the need for additional instruments that may make developers aware of the rationale behind the flakiness of UI tests. As such, our findings can (1) inform the designers of flaky test detection, diagnosing, and fixing techniques of the innate relation between UI-related issues and other, more fundamental problems causing flakiness; (2) call for more empirical investigations into the potential strategies and to manage multiple root causes of flakiness jointly.

Table 11 makes explicit the commonalities and differences between the results obtained in our work and those presented in previous research. With respect to Eck et al. [26], the major differences were observed in terms of the frequency of specific root causes. For instance, Eck et al. [26] reported that the most common root causes in Mozilla were due to *"Concurrency"*, *"Async Wait"*, and *"Too Restrictive Range"* issues, while our findings revealed *"API"* issues, *"Program Logic"*, and *"UI problems"* as the most common root causes in mobile apps. It is worth remarking that Eck et al. [26] analyzed an industrial context, i.e., Mozilla, and investigated test flakiness in web applications. On the contrary, we focused on mobile applications and targeted mobile practitioners. As such, the differences observed may be due to the *different context of the study*: our findings must be seen as complementary.

Concerning the work conducted by Ahmad et al. [2], the authors analyzed an industrial environment, hence collecting flakiness-related information from a different context. Similarly to the cases above, the major difference lies in the frequency of the root causes identified - Ahmad et al. [2] identified *"Async Wait"*, *"Test Order Dependency"*, and *"Randomness"* as the most frequent root causes, while our findings reported a different perspective on the matter when considering mobile apps. In the first place, the differences observed can be due to our *specific focus on mobile apps*: Ahmad et al. [2] did not explicitly report the types of applications they considered, yet we may suppose these were general-purpose and targeted different application domains. As a consequence, our work more deeply explores the problem of test flakiness in mobile applications. In the second place, the differences might also be connected to the *research methods employed*. Ahmad et al. [2] performed a qualitative analysis of the source code of the test suites of the closed-source systems taken into account, while we performed a mixed-method research to elicit the mobile practitioner's perception. Finally, another possible reason explaining the differences observed is concerned with the *number of practitioners* involved in the two studies: Ahmad et al. [2] recruited 18 developers and analyzed the test suites they developed, while we recruited 130 mobile developers.

The findings by Gruber and Fraser [38] first differ from ours in terms of the frequency of specific root causes. In particular, while the findings by Gruber and Fraser [38] reported *"Test Order Dependency"* as the most common root cause for both non-mobile developers and mobile developers, ours showed that *"API"* issues is the most common cause of flakiness in mobile apps, with emph"Test Order Dependency" only considered as the ninth most mentioned root cause. Still, in terms of root causes, the mobile developers surveyed by Gruber and Fraser [38] mentioned the use of emulators as a cause of increased frequency of flakiness, which our study did not confirm. Finally, concerning the developers' perception, Gruber and Fraser [38] suggested that flaky tests with a higher failure rate are perceived as more problematic, while our survey showed that the recurring emergence of the same flaky tests makes developers less and less inclined to address the causes of flakiness. In other terms, when com-

Table 11: Comparison of the findings between our study and the most closely related papers.

Related Work	Different Findings	Common Findings
Eck et al. [26]	**Differences**. Eck et al. [26] surveyed web developers, finding that Concurrency (26%), Async Wait (22%), and Too Restrictive Range (17%) are the most common root causes of flakiness. Our study surveyed mobile developers, finding that API issues (31%), Program Logic (30%), and UI problems (25%) are the most common root causes. **Possible rationale.** The different target of practitioners involved might explain the differences observed, along with our more specific focus on mobile apps.	• Prominence and relevance of test flakiness; • The RERUN approach is the most common technique used to detect flakiness.
Habchi et al. [43]	**Differences.** Habchi et al. [43] reported on how developers address test flakiness in the wild (e.g., building stable infrastructure), while we (i) proposed a catalog of more specific actions developers take when detecting the causes of test code flakiness in mobile apps; and (ii) reported on the fixing actions performed to deal with mobile test flakiness. **Possible rationale.** The different target of practitioners involved might explain the differences observed, along with our more specific focus on mobile apps. In addition, we surveyed a larger number of practitioners (130 vs 14).	• Prominence and relevance of test flakiness; • The RERUN approach is the most common technique used to detect flakiness.
Gruber and Fraser [38]	**Differences.** The authors reported Test Order Dependency as the most common root cause, while in our study, it appears only in 15% of the cases. In addition, the mobile developers surveyed by Gruber and Fraser [38] reported the use of emulators, while our practitioners did not mention this aspect. Different practitioners' perceptions. Gruber and Fraser [38] suggested that flaky tests with a higher failure rate are perceived as more problematic, while our study showed that the recurring emergence of the same flaky tests makes developers less and less inclined to address the causes of flakiness. **Possible rationale.** The different target of practitioners involved might explain the differences observed, along with our more specific focus on mobile apps. Gruber and Fraser [38] indeed considered a sample of general practitioners, while we specifically focused on the detection and fixing procedures applied by mobile developers.	• Prominence and relevance of test flakiness; • The RERUN approach is the most common technique used to detect flakiness; • The automated approaches to handle a flaky test are unknown or rarely used.
Ahmad et al. [2]	**Differences.** The study was conducted on closed-source test cases finding that Async Wait (91%), Test Order Dependency (5%), and Randomness (4%) are the most root causes of test flakiness. Ahmad et al. [2] did not specify the types of systems object of the study, hence we cannot speculate on the amount of mobile practitioners involved in the study. In any case, we discovered different root causes and fixing strategies that specifically target mobile applications. In addition, Ahmad et al. [2] analyzed the relation between test flakiness and factors such as robustness, age, and size. On the contrary, we assessed the relation between flaky tests and test smells. **Possible rationale.** The different research methods used to conduct the studies may explain the differences observed: Ahmad et al. [2] performed a qualitative analysis of the test suites of closed-source systems, while we performed mixed-method research to elicit the mobile practitioner's perception of test flakiness. Also, our specific focus on mobile apps might be an additional reason for the differences observed, along with the larger number of practitioners involved in our study (130 vs 18).	• Prominence and relevance of test flakiness.
Parry et al. [87]	**Differences.** Parry et al. [87] surveyed general-purpose developers, finding that Improper Setup and Teardown and Network-related Issues are the most common root causes of flakiness - they did not report the number of practitioners who indicated the root causes as frequent. Our study surveyed mobile developers, finding that API issues (31%), Program Logic (30%), and UI problems (25%) are the most common root causes. **Possible rationale.** The different target of practitioners involved might explain the differences observed, along with our more specific focus on mobile apps.	• Prominence and relevance of test flakiness.

pared to the paper by Gruber and Fraser [38], our work specifically characterized test code flakiness in mobile apps, providing an improved overview of the most problematic and frequent root causes of flakiness, hence emphasizing the need for treating test code flakiness differently when considering mobile apps. The differences observed might be due to two main reasons. In the first place, the *sample of practitioners*: Gruber and Fraser [38] surveyed 233 professionals from the general public all over the globe and 102 employees of the BMW Group, while our work targeted a sample of 130 mobile practitioners. In the second place, the *specificity of the analysis*: while Gruber and Fraser [38] aimed at collecting the root causes and practitioner's perception of test flakiness of general-purpose systems, we provided a more focused analysis of the mechanisms applied by mobile developers to deal with mobile test flakiness, also extending the current knowledge by investigating the specific fixing mechanisms they typically put in place. As a consequence, the findings of the two studies should be seen as complementary, since they provide different perspectives on the same matter by analyzing different contexts.

The work by Habchi et al. [43] focused on mapping the measures employed by practitioners when dealing with flaky tests. Our work is, instead, larger and more comprehensive, as we aimed to analyze (1) the prominence, (2) the root causes, and (3) the diagnosis and mitigation strategies applied by mobile developers when dealing with flakiness. In terms of findings, Habchi et al. [43] reported that developers address flaky tests by building stable infrastructures and enforcing guidelines: our work could further elaborate on the matter, proposing a catalog of more specific actions developers take when detecting and addressing the causes of test code flakiness. The *different scope* of the two studies, combined with the *different sample sizes* considered (14 versus 130 practitioners), may have led to different results, allowing us to extend the knowledge emerging from the paper by Habchi et al. [43].

Finally, the work by Parry et al. [87] surveyed 170 general-purpose practitioners recruited through social media channels. The goal of the study was to elicit the definition that practitioners provide for test flakiness, the impact of flaky tests, their root causes, and the actions performed to fix them. Parry et al. [87] identified *"Improper Setup and Teardown"* and *"Network-related Issues"* as the most common root causes. On the contrary, our work identified root causes that are intrinsically connected to the way mobile applications are developed, i.e., *"API"* issues, *"Program Logic"*, and *"UI problems"*. In other words, our work should be seen as complementary with respect to the work by Parry et al. [87]: the differences observed are likely to be due to the *different context of the study* and, as a consequence, to the *sample of practitioners*.

These findings recall the need for prioritizing in a different manner the research effort to spend to support the testing activities of mobile developers. Perhaps more importantly, one of the key results of our study concerns the different impacts that flaky tests have on the maintenance and evolution process. Most of the participants of the survey study indeed reported that they use to fix flaky tests when these are detected rather than disabling or ignoring them - this is in contrast with what has been widely reported in previous studies targeting traditional systems [26, 43]. Hence, our empirical analysis reveals that the socio-technical impact of flaky tests on the evolution of mobile apps is greater, highlighting that the managerial and technical strategies employed by mobile developers to evolve mobile apps might be affected by test flakiness.

6.2. Reflections, Implications, and Actionable Items

The findings obtained and the observations provided in the previous section, allow us to formulate a number of additional considerations and implications for both researchers and practitioners.

Test Flakiness: A Key Problem for Developers. While previous work pointed out the limited amount of testing activities performed by mobile developers [90], our study revealed that the problem of test flakiness is still perceived as relevant and problematic. Not only more than half of the surveyed participants have dealt with flaky tests at least once in their experience, but the problem seems to be more serious than in traditional software because of the peculiarities of mobile apps, like the intensive reliance on external APIs. This was also one of the main insights coming from the systematic grey literature review: especially when dealing with UI testing, developers might end up with flaky tests already when designing tests. Our study, therefore, represents a call for researchers working in the field of software testing: further empirical investigations on the current developer's practices, and the support practitioners need, other than on the nature and the impact of test flakiness, are critical missing pieces that our research community is called to address. In this sense, it is our hope that the results of the study and the additional implications discussed later in this section will inspire and stimulate further empirical and developer-centered research.

Root Causes of Flakiness: Beyond Test Code. Researchers have frequently investigated the sources of flakiness, identifying a number of test-related factors impacting the likelihood of tests being non-deterministic [17, 26, 72, 86]. Unfortunately, this is not enough or, at least, not fully generalizable to mobile apps. While developers recognized some of the known causes of flakiness, like concurrency, randomness, or test order dependency, they also put in the spotlight issues in production code design and in the verification of user interfaces. These represent key differences with respect to traditional software. For instance, application logic and API issues are not only the most diffused causes of flakiness according to our survey study but also those harder to detect and fix. While Thorve et al. [110] pioneered the analysis of how flaky tests manifest themselves in mobile apps, we advocate more research on the role played by the characteristics of production code that might increase the likelihood of tests becoming flaky. We can envision a number of further investigations into the impact of design-related aspects, like REST API design patterns and anti-patterns [10, 83], code smells [32, 112], usability patterns [80], and other usage patterns [53], on test code flakiness. We believe that new experimentations would also offer opportunities for joint research efforts among different research sub-communities in software engineering.

On the Role of Technical Debt. As a follow-up discussion point, it is important to further remark on the role that developers assigned to technical debt. As a side result of our analysis, we could indeed confirm that a number of test and code metrics and smells are perceived as connected to flakiness. This aspect has several implications. In the first place, some forms of technical debt, like the test smells capturing issues in the management of external resources, might act as proxy metrics that developers might monitor to keep the stability of test cases under

control. In this sense, additional studies on the characteristics making test cases smelly *and* flaky would be desirable. Furthermore, should the relation between test code design and flakiness be confirmed, this would open new interesting avenues for researchers working in predictive analytics [4, 92, 93] and software quality [55, 85, 91]: the former might be interested in assessing how well can test-related metrics and smells predict the emergence of flaky tests; the latter in devising novel mobile-specific instruments and analytical dashboards that could assist developers in monitoring and detecting potential sources of flakiness in advance. Last but not least, our results are interesting from the automated refactoring perspective. As a matter of fact, we still lack fully automated refactoring approaches to deal with test design concerns [36]. The relevance of these aspects on flakiness might boost researchers' willingness to invest effort on the matter.

Is the Automated Support Really Limited? Over the last years, researchers have been proposing several tools to deal with flakiness, especially when it turns to specific root causes like concurrency and test order dependency [86]. At the same time, our systematic grey literature review pointed out the availability of some automation mechanisms to ease the flaky test detection process. Nevertheless, developers do not seem to use (or even be aware of) the automated support available. We see a number of possible reasons behind this dichotomy. First, there seems to be a relevant gap between academia and industry, which should be addressed somehow. On the one hand, novel dissemination strategies might be experimented to understand how to better communicate research results to practitioners: the availability of practitioner's forums and/or blogs, e.g., REDDIT and MEDIUM, might represent a valuable starting point, i.e., the scientific community might consider the grey literature sources as a complementary advertisement method of dissemination. Second, our research community is called to devise more readily usable prototypes that would make academic techniques usable, allowing practitioners to more easily use or adopt them in practice. Our results further highlight the importance of devising additional policies to stimulate the publication of tools and frameworks developed in an academic setting. Last but not least, our findings might inspire educators, who might want to update the academic programs currently in place to include specific parts connected to the relation between academia and industry, other than letting the new generation of computer scientists get already exposed to academic prototypes, in the hope that this would then favor their larger adoption in the future.

Improving Test Code Review. One of the key findings of our study, even as a consequence of the previously discussed point, concerns the fact that developers mostly deal with flakiness manually. Our participants mentioned code review [6] as a useful instrument to diagnose and possibly treat flakiness in a collaborative fashion [89]—this is reasonable, given the popularity of the code review process [58]. To the best of our knowledge, however, research in this field is still neglected. Only a few researchers have proposed investigations in the context of test code review [106] or even attempted to understand how to best support developers during test code reviews [108]. Therefore, it is of the greatest urgency to build on this line of research and assess how current and future instruments might assist developers in the code review process.

On the Long-Term Effects of Flakiness. Our last discussion point relates to the findings achieved about the recurrence of flaky tests. Around 30% of developers claimed that test flakiness appears multiple times during software evolution. Besides the problems that this point may have on source code quality and effectiveness, we discovered that recurrent flakiness might have dramatic consequences for software testing, corroborating what was advocated by Melski [24] in a well-known blog post. Our participants indeed reported that the recurrence of flaky tests might lead them to decide to ignore, disable, or even remove the non-deterministic test case at the risk of missing relevant defects. Hence, the analysis of the evolutionary aspects of test flakiness represents a key challenge for researchers who are called to operate more in this direction.

7. Threats to Validity

Several confounding factors might have possibly influenced the results of our study.

Threats to Construct Validity. Threats in this category refer to the relationship between hypothesis and observations. To understand how mobile developers discuss test flakiness, we conducted a systematic grey literature review [35]. In this regard, possible threats refer to the soundness and completeness of the review. With respect to the former, the first two authors of this paper followed well-established guidelines to search, analyze, and select relevant sources; moreover, the joint work conducted by the two authors have reduced the risk of subjective evaluations of the resources to include as well as allowed a quick solving of possible disagreements. As for the latter, we analyzed all the relevant GOOGLE pages when gathering the study, also performing it using the incognito mode to avoid biases due to previous navigation history.

As for the survey study, these are mainly related to the way we designed the survey. Starting from the objectives posed, we attempted to define clear and explicit questions in the survey that could allow participants to properly understand the meaning/phrasing and provide an answer that could have been directly mapped onto our objectives. Also, whenever needed, we defined free-text answers to let developers express themselves freely, without restriction. The survey design involved all the paper's authors, who all have experience in software testing, empirical software engineering, and research design. While this already partially mitigated threats to construct validity, we also involved two external mobile software engineering developers in a pilot study. This additional investigation aimed at preliminarily assessing the opinions of our sample population, highlighting possible issues to be fixed before releasing the survey on a large scale. The pilot study let some minor concerns emerge that we promptly fixed. The follow-up double-checks of the involved developers let us be even more confident of the validity of our survey. Nonetheless, replications of the survey study would be beneficial to discover additional points of view and perspectives that we might have missed. In this respect, we made all our data and scripts publicly available to make our results repeatable and reproducible [95].

Threats to Internal Validity. The recruitment strategy employed in our study might have led to the selection of a biased sample. This is especially true since we relied on voluntary participation through an online instru-

ment like PROLIFIC. In this respect, there are three considerations to make. First, involving developers is always challenging: this is also visible from the relatively low response rate of survey studies in software engineering [56, 71, 96]. As such, accepting the limitations coming from the recruitment through online platforms is often the only means to conduct these studies. In addition, we mitigated possible self-selection or voluntary response bias [48, 100] by providing participants with a payment as incentive. This incentive aims to stimulate broader participation, hence reducing threats to validity. Last but not least, our results corroborate some of the findings achieved in previous studies [26, 72], hence increasing the confidence in the validity of the insights we reported. However, we cannot still exclude possible influences of the recruitment strategy on our results. On the one hand, we plan to perform additional studies on test flakiness in mobile applications: these will have the goal of triangulating the results of our survey study. On the other hand, industrial replications or further experimentation on the developer's opinions about flaky tests in mobile apps would be beneficial to corroborate our findings.

Threats to Conclusion Validity. As for the potential threats to the conclusions drawn, the data analysis methods are worth discussing. In the systematic grey literature review, we conducted content analysis sessions to label and summarize the themes in the selected sources. To reduce the potential subjectivity of the analysis, two of the authors acted as inspectors, defining a formal protocol to assess the suitability of each resource for the research questions of the study and opening a discussion on the labels and descriptions assigned to each source.

Another discussion point concerns the fact that the practitioners involved in the study—both those writing on the grey literature sources and those who participated in the survey study—might not be mobile developers only and, therefore, might have expressed opinions that are not necessarily specific to the mobile context. Being aware of this potential limitation, we performed quality assessment checks to verify the suitability of the sources considered with respect to the research questions of the study. In addition, we include questions in the survey that explicitly asked participants to answer, keeping in mind that the problem of test flakiness should have been assessed within the context of mobile apps. In some cases, e.g., when considering the diffuseness of the root causes of flakiness, we also requested participants to provide their opinion by explicitly comparing mobile and non-mobile environments: in this way, the participants were stimulated to reason on the peculiarities of flaky tests in the two different contexts, hence providing us with more reliable insights into the problem.

As for the survey study, we used statistical methods to interpret numeric distributions coming from the responses of the participants. For instance, we employed frequency analysis to describe the most common root causes reported by the involved practitioners. At the same time, we did not use additional statistical methods and tests, e.g., we did not compare the numeric distributions pertaining to two questions of the survey to find out statistically significant differences. This was done on purpose, as the distributions coming from the responses relate to different concepts and address different perspectives of the test flakiness problem. Statistical comparisons would not apply in this case. For the open questions, we had to conduct content analysis to interpret the opinions of developers. In this respect, we systematically approached the matter with two experienced researchers

involved. In any case, we released all the material produced in the context of this study to enable the verifiability of the conclusions described in our online appendix [95].

An additional point of discussion concerns the data cleaning process conducted, who might have missed the exclusion of answers clearly released by participants who did not take the task seriously or did not have enough expertise to approach our survey. In this respect, the paper's first author went through each response and validated it. Then, a second author confirmed the operations performed. Such a double-check makes us confident of not having considered answers that might have biased our conclusions.

Another possible threat to the validity of our study is related to the reliability of the opinions collected by surveying practitioners. In this respect, previous studies showed that developers are sometimes inaccurate when reporting on the amount of testing they perform [13, 14]. Similarly, they might be inaccurate when reporting additional information concerned with the activities performed to deal with test flakiness. On the one hand, we made sure to profile the participants by inquiring about their experience and the testing practices they performed. The self-reported information of our sample was in line with the quantitative findings by Pecorelli et al. [90] in terms of amount of testing performed: such a congruence may indicate that the participants approached our study in a proactive manner, providing insights that well represent their daily activities. On the other hand, other researchers have approached the problem of flaky tests through survey studies [43, 87], reporting compelling evidence of the validity of this research method to understand the properties of the test flakiness problem.

Threats to External Validity. Threats in this category are concerned with the generalizability of the results. When performing the systematic grey literature review, there is a risk of missing relevant resources because concepts related to those included in our search strings are differently named in such studies. Some studies may refer to non-deterministic tests instead of test flakiness. To mitigate this, we have explicitly included all relevant synonyms and similar words in our search strings. We have also exploited the features offered by search engines, which naturally support considering related terms for all those contained in a search string. Items found using the search terms have been assessed thoroughly based on various dimensions of quality [35]. Despite our efforts, we still identified a few amounts of resources. This limits the generalizability of the findings and the representativeness of the sample considered. For this reason, future replications of the grey literature review would be desirable and might lead to different results with respect to those reported in our paper.

As for the survey study, the conclusions pertain to a specific sample of 130 developers having the characteristics discussed in Section 5.4. We noticed that the survey participants were pretty heterogeneous in terms of working contexts: indeed, freelance, industrial, open-source, and startup developers composed the 36.6%, 35.1%, 28.2%, and 26.1% of our sample, respectively. On the contrary, we did not observe much heterogeneity when considering the other background questions—this was already perceivable in Figure 3, which showed how the background of participants is somehow similar in terms of years of experience as mobile app developers, type of developers, and development in CI/CD environment. On the basis of these observations we decided to further test

our conclusions, elaborating on our findings with respect to the different working contexts. Hence, we conducted an additional analysis of the responses collected.

Specifically, we first group the responses by working context. In this way, we could analyze the responses coming from open-source, industrial, startup, and freelance developers individually, in an effort to analyze whether the developers of the various clusters reported similar considerations and data. We excluded the responses given by developers who qualified themselves as a combination of the four categories, e.g., we excluded the responses of developers who were both open-source and industrial developers - this was needed to let us observe the potential differences among the various clusters in isolation. For each cluster, we then repeated the data analysis and compared the observations obtained.

For the questions whose answer was given through a 5-point Likert scale, we could compare the responses in the four clusters through the Mann-Whitney test [79], which is a non-parametric statistical test that verifies whether randomly selected values from two distributions belong to the same statistical population—in our case, the test allowed us to assess whether the responses provided by developers of different clusters were not statistically different. For the multiple-choice questions, we compared the responses by computing percentages over the values assigned by respondents, e.g., we computed how many times open-source developers declared to have faced flaky tests very often and compared the percentage to the one computed on the set of responses given by industrial developers. Finally, for the open questions we verified whether the content analysis sessions originally conducted by the first two authors of the paper (see Section 5.3) revealed differences in terms of concepts expressed by developers in the clusters.

The additional analysis did not reveal major variations among the clusters—detailed results are reported in our online appendix [95]. The Mann-Whitney test reported that the 5-point answers were not statistically different in any case. Also, the percentages computed on multiple-choice questions were similar, hence highlighting that similar responses were provided by developers independently from the working context. In addition, the concepts expressed within the open questions pointed out similar considerations.

In conclusion, our analysis reveals no significant differences among the responses. While we are not aware of any reliable statistical report showing how our distribution of participants per working context compares to the largest population of mobile developers, the absence of significant differences among the answers provided by the developers in our sample increases our confidence in the generalizability of our findings. Yet, replications of the study would still be welcome to reinforce our results and increase the knowledge of how test flakiness is perceived and managed in different contexts.

Still, in terms of generalizability, 79% of our participants develop for the ANDROID operating system. While this skews our sample toward the population of ANDROID developers, it is also worth pointing out that our sample is in line with respect to the largest population of mobile developers. Indeed, according to a recent report released by a

well-known and independent statistical corporation such as STATISTA,[9] the large majority of mobile practitioners develop for the ANDROID operating system, while other platforms are significantly less targeted. Nonetheless, we acknowledge such a limitation and remark that further replications of our study would be desirable and part of our future research agenda.

8. Conclusion

In this paper, we analyzed the mobile developer's perspective on the test flakiness problem. Using a mixed-method research approach, we combined the outcomes of a systematic grey literature review with those of a survey study to address three research questions concerned with (1) the prominence and harmfulness of flaky tests; (2) the most common root causes of flakiness; and (3) the current processes applied to diagnose and fix flaky tests. According to our findings, we claim that the problem of test flakiness should be further explored by the research community since it is perceived as prominent and harmful by developers. In addition, the analysis of the current methods employed to deal with test flakiness let emerge several limitations that would be worth to further investigating. Last but not least, our study indicated that the presence of flaky tests might be correlated to various characteristics of source code, e.g., test smells or external APIs, and is more problematic when it comes to the verification of user interfaces.

To sum up, the main contributions that our work has brought to the community are the following:

1. The first systematic grey literature review on test flakiness in mobile applications, which allowed for verification of the prominence of flaky tests and the practices employed by practitioners to handle them;

2. A survey study that involved 130 mobile developers and that allowed us to corroborate and extend the results of the systematic grey literature review;

3. A publicly available online appendix [95] reporting all the (anonymous) data collected during our analyses and that can be exploited by further researchers to build on top of our findings.

The implications of the study provide the research community with a number of research challenges and opportunities that we hope might be helpful to let the problem of test flakiness in mobile applications be more and more explored in the future. Novel empirical studies targeting specific aspects that emerged from our study (e.g., the relation between flakiness and API issues), other than the definition of automated techniques to support developers when dealing with flaky tests, are part of our future research agenda.

[9] The STATISTA report on the distribution of mobile developers by operating system: https://www.statista.com/statistics/1078678/software-development-operating-system-mobile/.

Acknowledgement

Fabio is partially funded by the Swiss National Science Foundation through the SNF Projects No. PZ00P2_-186090. This work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript has data included as electronic supplementary material. In particular: resources analyzed in the context of grey literature and survey study conducted, detailed results, as well as scripts and additional resources useful for reproducing the study, are available as part of our online appendix on Figshare: https://doi.org/10.6084/m9.figshare.24183279.

Credits

Valeria Pontillo: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Fabio Palomba**: Conceptualization, Methodology, Validation, Writing - Review & Editing. **Filomena Ferrucci**: Supervision, Resources, Writing - Review & Editing.

Bibliography

[1] Abdellatif, M., Tighilt, R., Belkhir, A., Moha, N., Guéhéneuc, Y., Beaudry, É., 2020. A multi-dimensional study on the state of the practice of rest apis usage in Android apps. Automated Software Engineering 27, 187–228.

[2] Ahmad, A., Leifler, O., Sandahl, K., 2021. Empirical analysis of practitioners' perceptions of test flakiness factors. Software Testing, Verification and Reliability 31, e1791.

[3] Alrubaye, H., Alshoaibi, D., Alomar, E., Mkaouer, M.W., Ouni, A., 2020. How does library migration impact software quality and comprehension? an empirical study, in: International Conference on Software and Software Reuse, Springer. pp. 245–260.

[4] Alshammari, A., Morris, C., Hilton, M., Bell, J., 2021. Flakeflagger: Predicting flakiness without rerunning tests, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 1572–1584.

[5] Andrews, D., Nonnecke, B., Preece, J., 2007. Conducting research on the internet: Online survey design, development and implementation guidelines .

[6] Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE. pp. 712–721.

[7] Bakker, C., Wang, F., Huisman, J., Den Hollander, M., 2014. Products that go round: exploring product life extension through design. Journal of cleaner Production 69, 10–16.

[8] Barboni, M., Bertolino, A., De Angelis, G., 2021. What we talk about when we talk about software test flakiness, in: International Conference on the Quality of Information and Communications Technology, Springer. pp. 29–39.

[9] Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C.E., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2014. The impact of api change-and fault-proneness on the user ratings of Android apps. IEEE Transactions on Software Engineering 41, 384–407.

[10] Belkhir, A., Abdellatif, M., Tighilt, R., Moha, N., Guéhéneuc, Y., Beaudry, É., 2019. An observational study on the state of rest api uses in Android mobile applications, in: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE. pp. 66–75.

[11] Bell, J., Kaiser, G., Melski, E., Dattatreya, M., 2015. Efficient dependency detection for safe Java test acceleration, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 770–781.

[12] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D., 2018. DeFlaker: Automatically detecting flaky tests, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE. pp. 433–444.

[13] Beller, M., Gousios, G., Panichella, A., Zaidman, A., 2015a. When, how, and why developers (do not) test in their ides, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 179–190.

[14] Beller, M., Gousios, G., Zaidman, A., 2015b. How (much) do developers test?, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE. pp. 559–562.

[15] Benzies, K.M., Premji, S., Hayden, K.A., Serrett, K., 2006. State-of-the-evidence reviews: advantages and challenges of including grey literature. Worldviews on Evidence-Based Nursing 3, 55–61.

[16] Bielova, N., Dragoni, N., Massacci, F., Naliuka, K., Siahaan, I., 2009. Matching in security-by-contract for mobile code. The Journal of Logic and Algebraic Programming 78, 340–358.

[17] Camara, B., Silva, M., Endo, A., Vergilio, S., 2021. On the use of test smells for prediction of flaky tests, in: Brazilian Symposium on Systematic and Automated Software Testing, pp. 46–54.

[18] Cavanagh, S., 1997. Content analysis: concepts, methods and applications. Nurse researcher 4, 5–16.

[19] Cordella, M., Alfieri, F., Clemm, C., Berwald, A., 2021. Durability of smartphones: A technical analysis of reliability and repairability aspects. Journal of Cleaner Production 286, 125388.

[20] Cordy, M., Rwemalika, R., Franci, A., Papadakis, M., Harman, M., 2022. Flakime: laboratory-controlled test flakiness impact assessment, in: Proceedings of the 44th International Conference on Software Engineering, pp. 982–994.

[21] Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., De Lucia, A., 2017. Software-based energy profiling of Android apps: Simple, efficient and reliable?, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, IEEE. pp. 103–114.

[22] Di Sorbo, A., Panichella, S., 2021. Exposed! a case study on the vulnerability-proneness of google play apps. Empirical Software Engineering 26, 1–31.

[23] Dong, Z., Tiwari, A., Yu, X.L., Roychoudhury, A., 2021. Flaky test detection in Android via event order exploration, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 367–378.

[24] E., M., 2020. 6 tips for writing robust, maintainable unit tests. https://blog.melski.net/tag/unit-tests/.

[25] Ebert, F., Serebrenik, A., Treude, C., Novielli, N., Castor, F., 2022. On recruiting experienced github contributors for interviews and surveys on prolific, in: International Workshop on Recruiting Participants for Empirical Software Engineering.

[26] Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A., 2019. Understanding flaky tests: The developer's perspective, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 830–840.

[27] Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments, in: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 235–245.

[28] Fatima, S., Ghaleb, T.A., Briand, L., 2022. Flakify: A black-box, language model-based predictor for flaky tests. IEEE Transactions on

Software Engineering .

[29] Fazzini, M., Orso, A., 2017. Automated cross-platform inconsistency detection for mobile apps, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 308–318.

[30] Ferrer, J., Chicano, F., Alba, E., 2013. Estimating software testing complexity. Information and Software Technology 55, 2125–2139.

[31] Fowler, M., 2011. Eradicating non-determinism in tests. Martin Fowler Personal Blog .

[32] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

[33] Francese, R., Gravino, C., Risi, M., Scanniello, G., Tortora, G., 2017. Mobile app development and management: results from a qualitative investigation, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE. pp. 133–143.

[34] Gambi, A., Bell, J., Zeller, A., 2018. Practical test dependency detection, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 1–11.

[35] Garousi, V., Felderer, M., Mäntylä, M.V., 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information and Software Technology 106, 101–121.

[36] Garousi, V., Küçük, B., 2018. Smells in software test code: A survey of knowledge in industry and academia. Journal of Systems and Software 138, 52–81.

[37] Greca, R., Miranda, B., Bertolino, A., 2023. Orchestration strategies for regression test suites, in: 2023 IEEE/ACM International Conference on Automation of Software Test (AST), IEEE. pp. 163–167.

[38] Gruber, M., Fraser, G., 2022. A survey on how test flakiness affects developers and what support they need to address it, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 82–92.

[39] Gruber, M., Heine, M., Oster, N., Philippsen, M., Fraser, G., 2023. Practical flaky test prediction using common code evolution and test history data, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 210–221.

[40] Gruber, M., Lukasczyk, S., Kroiß, F., Fraser, G., 2021. An empirical study of flaky tests in Python, in: 2021 14th IEEE Conference on Software Testing, Verification and Validation, IEEE. pp. 148–158.

[41] Gyori, A., Shi, A., Hariri, F., Marinov, D., 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency, in: 2015 International Symposium on Software Testing and Analysis, pp. 223–233.

[42] Habchi, S., Cordy, M., Papadakis, M., Traon, Y.L., 2021. On the use of mutation in injecting test order-dependency. arXiv preprint arXiv:2104.07441 .

[43] Habchi, S., Haben, G., Papadakis, M., Cordy, M., Le Traon, Y., 2022a. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 244–255.

[44] Habchi, S., Haben, G., Sohn, J., Franci, A., Papadakis, M., Cordy, M., Le Traon, Y., 2022b. What made this test flake? pinpointing classes responsible for test flakiness, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 352–363.

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

[46] Hall, T., Flynn, V., 2001. Ethical issues in software engineering research: a survey of current practice. Empirical Software Engineering 6, 305–317.

[47] Hashemi, N., Tahir, A., Rasheed, S., 2022. An empirical study of flaky tests in Javascript, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 24–34.

[48] Heckman, J.J., 1990. Selection bias and self-selection, in: Econometrics. Springer, pp. 201–224.

[49] Holl, K., Elberzhager, F., 2016. Mobile application quality assurance: Reading scenarios as inspection and testing support, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE. pp. 245–249.

[50] Hunt, K.J., Shlomo, N., Addington-Hall, J., 2013. Participant recruitment in sensitive surveys: a comparative trial of 'opt in'versus 'opt out'approaches. BMC Medical Research Methodology 13, 1–8.

[51] Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., 2019. Formal methods for Android banking malware analysis and detection, in: 2019

Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), IEEE. pp. 331–336.

[52] Jabangwe, R., Edison, H., Duc, A.N., 2018. Software engineering process models for mobile app development: A systematic literature review. Journal of Systems and Software 145, 98–111.

[53] Jin, Y., Duffield, N., Gerber, A., Haffner, P., Hsu, W., Jacobson, G., Sen, S., Venkataraman, S., Zhang, Z., 2012. Characterizing data usage patterns in a large cellular network, in: 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design, pp. 7–12.

[54] Joorabchi, M.E., Mesbah, A., Kruchten, P., 2013. Real challenges in mobile app development, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE. pp. 15–24.

[55] Kim, D.J., Chen, T.P., Yang, J., 2021. The secret life of test smells-an empirical study on test smell evolution and maintenance. Empirical Software Engineering 26, 1–47.

[56] Kitchenham, B.A., Pfleeger, S.L., 2002. Principles of survey research part 2: designing a survey. ACM SIGSOFT Software Engineering Notes 27, 18–20.

[57] Klayman, J., 1995. Varieties of confirmation bias. Psychology of learning and motivation 32, 385–418.

[58] Kononenko, O., Baysal, O., Godfrey, M.W., 2016. Code review quality: How developers see it, in: ACM/IEEE 38th International Conference on Software Engineering, pp. 1028–1038.

[59] Kotarba, M., 2017. Measuring digitalization: Key metrics. Foundations of Management 9, 123–138.

[60] Kumara, I., Garriga, M., Romeu, A.U., Di Nucci, D., Palomba, F., Tamburri, D.A., van den Heuvel, W.J., 2021. The do's and don'ts of infrastructure code: A systematic gray literature review. Information and Software Technology 137, 106593.

[61] Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S., 2019a. Root causing flaky tests in a large-scale industrial setting, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 101–111.

[62] Lam, W., Muşlu, K., Sajnani, H., Thummalapenta, S., 2020a. A study on the lifecycle of flaky tests, in: ACM/IEEE 42nd International Conference on Software Engineering, pp. 1471–1482.

[63] Lam, W., Oei, R., Shi, A., Marinov, D., Xie, T., 2019b. iDFlakies: A framework for detecting and partially classifying flaky tests, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE. pp. 312–322.

[64] Lam, W., Winter, S., Astorga, A., Stodden, V., Marinov, D., 2020b. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects, in: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), IEEE. pp. 403–413.

[65] Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., Bell, J., 2020c. A large-scale longitudinal study of flaky tests. Proceedings of the ACM on Programming Languages 4, 1–29.

[66] Lampel, J., Just, S., Apel, S., Zeller, A., 2021. When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla, in: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1381–1392.

[67] Li, C., Khosravi, M.M., Lam, W., Shi, A., 2023. Systematically producing test orders to detect order-dependent flaky tests, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 627–638.

[68] Li, C., Zhu, C., Wang, W., Shi, A., 2022. Repairing order-dependent flaky tests via test generation, in: Proceedings of the 44th International Conference on Software Engineering, pp. 1881–1892.

[69] Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., Guéhéneuc, Y., 2014. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps, in: 22nd International Conference on Program Comprehension, pp. 232–243.

[70] Linares-Vasquez, M., Vendome, C., Luo, Q., Poshyvanyk, D., 2015. How developers detect and fix performance bottlenecks in Android apps, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 352–361.

[71] Lo, D., Nagappan, N., Zimmermann, T., 2015. How practitioners perceive the relevance of software engineering research, in: 10th Joint Meeting on Foundations of Software Engineering, pp. 415–425.

[72] Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests, in: Proceedings of the 22nd ACM SIGSOFT Interna-

tional Symposium on Foundations of Software Engineering, pp. 643–653.

[73] Martin, W., Sarro, F., Jia, Y., Zhang, Y., Harman, M., 2016. A survey of app store analysis for software engineering. IEEE Transactions on Software Engineering 43, 817–847.

[74] Memon, A., Cohen, M., 2013. Automated testing of GUI applications: models, tools, and controlling flakiness, in: International Conference on Software Engineering (ICSE'13), IEEE. pp. 1479–1480.

[75] Micco, J., 2017. The state of continuous integration testing@ google .

[76] Morán, J., Augusto, C., Bertolino, A., de la Riva, C., Tuya, J., 2019. Debugging flaky tests on web applications., in: WEBIST, pp. 454–461.

[77] Morán Barbón, J., Augusto Alonso, C., Bertolino, A., Riva Álvarez, C.A., Tuya González, P.J., et al., 2020. Flakyloc: flakiness localization for reliable test suites in web applications. Journal of Web Engineering, 2 .

[78] Morin, K.H., 2013. Value of a pilot study.

[79] Nachar, N., et al., 2008. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. Tutorials in quantitative Methods for Psychology 4, 13–20.

[80] Nayebi, F., Desharnais, J.M., Abran, A., 2012. The state of the art of mobile application usability evaluation, in: 2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), IEEE. pp. 1–4.

[81] Nayebi, M., Adams, B., Ruhe, G., 2016. Release practices for mobile apps–what do users and developers think?, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE. pp. 552–562.

[82] Nemoto, T., Beglar, D., 2014. Likert-scale questionnaires, in: JALT 2013 conference proceedings, pp. 1–8.

[83] Oumaziz, M.A., Belkhir, A., Vacher, T., Beaudry, E., Blanc, X., Falleri, J., Moha, N., 2017. Empirical study on rest apis usage in Android mobile applications, in: International Conference on Service-Oriented Computing, Springer. pp. 614–622.

[84] Palomba, F., Linares-Vásquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2018a. Crowdsourcing user reviews to support the evolution of mobile apps. Journal of Systems and Software 137, 143–162.

[85] Palomba, F., Zaidman, A., De Lucia, A., 2018b. Automatic test smell detection using information retrieval techniques, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 311–322.

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

[87] Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2022. Surveying the developer experience of flaky tests, in: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, pp. 253–262.

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

[89] Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. ACM on Human-Computer Interaction 2, 1–27.

[90] Pecorelli, F., Catolino, G., Ferrucci, F., De Lucia, A., Palomba, F., 2022. Software testing and Android applications: a large-scale empirical study. Empirical Software Engineering 27, 1–41.

[91] Peruma, A., Almalki, K., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F., 2020. Tsdetect: An open source test smells detection tool, in: 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1650–1654.

[92] Pinto, G., Miranda, B., Dissanayake, S., d'Amorim, M., Treude, C., Bertolino, A., 2020. What is the vocabulary of flaky tests?, in: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 492–502.

[93] Pontillo, V., Palomba, F., Ferrucci, F., 2021. Toward static test flakiness prediction: A feasibility study, in: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, pp. 19–24.

[94] Pontillo, V., Palomba, F., Ferrucci, F., 2022a. Static test flakiness prediction: How far can we go? Empirical Software Engineering 27, 1–44.

[95] Pontillo, V., Palomba, F., Ferrucci, F., 2022b. Test code flakiness in mobile apps: The developer's perspective. https://doi.org/10.6084/m9.figshare.24183279.

[96] Punter, T., Ciolkowski, M., Freimut, B., John, I., 2003. Conducting on-line surveys in software engineering, in: 2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings., IEEE. pp. 80–88.

[97] Rehman, M.H.U., Rigby, P.C., 2021. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson, in: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1371–1380.

[98] Reid, B., Wagner, M., d'Amorim, M., Treude, C., 2022. Software engineering user study recruitment on prolific: An experience report. arXiv preprint arXiv:2201.05348 .

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

[100] Sakshaug, J.W., Schmucker, A., Kreuter, F., Couper, M.P., Singer, E., 2016. Evaluating active (opt-in) and passive (opt-out) consent bias in the transfer of federal contact data to a third-party survey agency. Journal of Survey Statistics and Methodology 4, 382–416.

[101] Salza, P., Palomba, F., Di Nucci, D., De Lucia, A., Ferrucci, F., 2020. Third-party libraries in mobile apps. Empirical Software Engineering 25, 2341–2377.

[102] Sarro, F., Harman, M., Jia, Y., Zhang, Y., 2018. Customer rating reactions can be predicted purely using app features, in: 2018 IEEE 26th International Requirements Engineering Conference (RE), IEEE. pp. 76–87.

[103] Shi, A., Bell, J., Marinov, D., 2019a. Mitigating the effects of flaky tests on mutation testing, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 112–122.

[104] Shi, A., Lam, W., Oei, R., Xie, T., Marinov, D., 2019b. ifixflakies: A framework for automatically fixing order-dependent flaky tests, in: 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 545–555.

[105] Silva, D., Teixeira, L., d'Amorim, M., 2020. Shake it! detecting flaky tests caused by concurrency with shaker, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 301–311.

[106] Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., Bacchelli, A., 2019. Test-driven code review: an empirical study, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 1061–1072.

[107] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A., 2018. On the relation of test smells to software code quality, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 1–12.

[108] Subramanian, S.V.V., McIntosh, S., Adams, B., 2020. Quantifying, characterizing, and mitigating flakily covered program elements. IEEE Transactions on Software Engineering .

[109] Terragni, V., Salza, P., Ferrucci, F., 2020. A container-based infrastructure for fuzzy-driven root causing of flaky tests, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), IEEE. pp. 69–72.

[110] Thorve, S., Sreshtha, C., Meng, N., 2018. An empirical study of flaky tests in Android apps, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 534–538.

[111] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells, in: 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 4–15.

[112] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2017. When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43, 1063–1088.

[113] Vahabzadeh, A., Fard, A.M., Mesbah, A., 2015. An empirical study of bugs in test code, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 101–110.

[114] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V., 2015. Quality and productivity outcomes relating to continuous integration in github, in: 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 805–816.

[115] Verdecchia, R., Cruciani, E., Miranda, B., Bertolino, A., 2021. Know you neighbor: Fast static prediction of test flakiness. IEEE Access 9, 76119–76134.

[116] Wang, R., Chen, Y., Lam, W., 2022. iPFlakies: A framework for detecting and fixing Python order-dependent flaky tests, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 120–124.

[117] Wasserman, A.I., 2010. Software engineering issues for mobile application development, in: FSE/SDP Workshop on Future of Software Engineering Research, pp. 397–400.

[118] Wei, A., Yi, P., Li, Z., Xie, T., Marinov, D., Lam, W., 2022. Preempting flaky tests via non-idempotent-outcome tests, in: Proceedings of the 44th International Conference on Software Engineering, pp. 1730–1742.

[119] Wessel, M., Gerosa, M.A., Shihab, E., 2022. Software bots in software engineering: benefits and challenges, in: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 724–725.

[120] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in software engineering. Springer Science & Business Media.

[121] Wu, Z., Jiang, Y., Liu, Y., Ma, X., 2020. Predicting and diagnosing user engagement with mobile UI animation via a data-driven approach, in: 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–13.

[122] Zhang, H., Zhou, X., Huang, X., Huang, H., Babar, M.A., 2020. An evidence-based inquiry into the use of grey literature in software engineering, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE. pp. 1422–1434.

[123] Zhang, J., Sagar, S., Shihab, E., 2013. The evolution of mobile apps: An exploratory study, in: Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, pp. 1–8.

[124] Zheng, W., Liu, G., Zhang, M., Chen, X., Zhao, W., 2021. Research progress of flaky tests, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 639–646.

[125] Ziftci, C., Cavalcanti, D., 2020. De-flake your tests: Automatically locating root causes of flaky tests in code at google, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 736–745.

[126] Zolfaghari, B., Parizi, R.M., Srivastava, G., Hailemariam, Y., 2021. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. Software: Practice and Experience 51, 851–867.