The Journal of Systems & Software 206 (2023) 111837

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Test flakiness' causes, detection, impact and responses: A multivocal review[☆]

Amjed Tahir[a,*,1], Shawn Rasheed[b,1], Jens Dietrich[c], Negar Hashemi[a], Lu Zhang[d]

[a] School of Mathematical and Computational Sciences, Massey University, New Zealand
[b] Information & Communication Technology Group, UCOL - Te Pūkenga, New Zealand
[c] School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
[d] Key Laboratory of High Confidence Software Technologies, MoE and the Department of Computer Science and Technology, Peking University, China

ARTICLE INFO

Article history:
Received 28 July 2022
Received in revised form 3 May 2023
Accepted 4 September 2023
Available online 7 September 2023

Keywords:
Flaky tests
Non-deterministic tests
Test bugs
Software testing
Multivocal review

ABSTRACT

Flaky tests (tests with non-deterministic outcomes) pose a major challenge for software testing. They are known to cause significant issues, such as reducing the effectiveness and efficiency of testing and delaying software releases. In recent years, there has been an increased interest in flaky tests, with research focusing on different aspects of flakiness, such as identifying causes, detection methods and mitigation strategies. Test flakiness has also become a key discussion point for practitioners (in blog posts, technical magazines, etc.) as the impact of flaky tests is felt across the industry. This paper presents a *multivocal review* that investigates how flaky tests, as a topic, have been addressed in both research and practice. Out of 560 articles we reviewed, we identified and analysed a total of 200 articles that are focused on flaky tests (composed of 109 academic and 91 grey literature articles/posts) and structured the body of relevant research and knowledge using four different dimensions: *causes*, *detection*, *impact* and *responses*. For each of those dimensions, we provide categorization and classify existing research, discussions, methods and tools With this, we provide a comprehensive and current snapshot of existing thinking on test flakiness, covering both academic views and industrial practices, and identify limitations and opportunities for future research.

1. Introduction

Software testing is a standard method used to uncover defects. Developers use tests early during development to uncover software defects when corrective actions are relatively inexpensive. A test can only provide useful feedback if it has the same outcome (either pass or fail) for every execution with the same code version. Tests with non-deterministic outcomes (known as *flaky tests*) may pass in some runs and fail in others. Such flaky behaviour is problematic as it leads to uncertainty in choosing corrective measures (Harman and O'Hearn, 2018). They also incur heavy costs in developers' time and other resources, particularly when the test suites are large and the development follows an agile methodology, requiring frequent regression testing on code changes to safeguard releases.

Test flakiness has been attracting more attention in recent years. In particular, there are several studies on the causes and impact of flaky tests in open-source and proprietary software. A study of open source projects observed that 13% of failed builds are due to flaky tests (Labuschagne et al., 2017). At Google, it was reported that around 16% of their tests were flaky, and 1 in 7 of the tests written by their engineers occasionally fail in a way that is not caused by changes to the code or tests (Micco, 2016). GitHub also reported that, in 2020, one in eleven commits (9%) had at least one red build caused by a flaky test (Raine, 2020). Other industrial reports have shown that flaky tests present a real problem in practice that have a wider impact on product quality and delivery (Fowler, 2011b; Sandhu, 2015; Palmer, 2019). Studies of test flakiness have also been covered in the context of several programming languages, including Java (Luo et al., 2022), Python (Gruber et al., 2021b) and, more recently, JavaScript (Hashemi et al., 2022).

Awareness that more research on test flakiness is needed has increased in recent years (Harman and O'Hearn, 2018). Currently, studies on test flakiness and its causes largely focus on specific sources of test flakiness, such as order-dependency (Gambi et al., 2018b), concurrency (Dong et al., 2021c), or UI-specific flakiness (Memon and Cohen, 2013; Romano et al., 2021). Given that test flakiness is an issue in both research and practice, we deem it important to integrate knowledge about flaky tests from both academic literature and grey literature in order to provide insights into the state of the practice.

[☆] Editor: Dr Burak Turhan.

[*] Corresponding author.
E-mail address: a.tahir@massey.ac.nz (A. Tahir).
[1] Joint first authors

https://doi.org/10.1016/j.jss.2023.111837

To address this, we performed a multivocal literature review on flaky tests. A multivocal review is a form of a *systematic literature review* (Kitchenham and Charters, 2007), which includes sources from both academic (formal) and grey literature (Garousi et al., 2019). Such reviews in computer science and software engineering have become popular over the past few years (Tom et al., 2013; Garousi and Küçük, 2018; Islam et al., 2019; Butijn et al., 2020) as it is acknowledged that the majority of developers and practitioners do not publish their work or thoughts through peer-reviewed academic channels (Garousi et al., 2016; Glass and DeMarco, 2006), but rather in blogs, discussion boards and Q&A sites (Williams, 2019).

This research summarizes existing work and current thinking on test flakiness from academic and grey literature. We hope that this can help a reader to develop an in-depth understanding of common causes of test flakiness, methods used to detect flaky tests, strategies used to avoid and eliminate them, and the impact flaky tests have. We identify current challenges and suggest possible future directions for research in this area.

The remaining part of the paper is structured as follows: Section 2 presents recent reviews and surveys on the topic. Our review methodology is explained in Section 3. We present our results by answering all four research questions in Section 4, followed by a discussion of the results in Section 5. Threats to validity are presented in Section 6, and finally, we present our conclusion in Section 7.

2. Related work

There has been a growing interest in flaky tests in recent years, especially after the publication of Martin Fowler's article on the potential issues with non-deterministic tests (Fowler, 2011b), and Luo et al.'s (Luo et al., 2022) seminal study.

To the best of our knowledge, there have been three reviews of studies on test flakiness: two systematic literature reviews, one by Zolfaghari et al. and the other by Zolfaghari et al. (2020) and Zheng et al. (2021) and a survey by Parry et al. (2021).

There have also been some developers' surveys that aimed to understand how developers perceive and deal with flaky tests in practice. A developer survey conducted by Eck et al. (2019b) with 21 Mozilla developers studied the nature and the origin of 200 flaky tests that had been fixed by the same developers. The survey looked into how those tests were introduced and fixed, and found that there are 11 main causes for those 200 flaky tests (including concurrency, async wait and test order dependency). It was also pointed out that flaky tests can be the result of issues in the production code (code under test) rather than in the test. The authors also surveyed another 121 developers about their experience with flaky tests. It was found that flakiness is perceived as a significant issue by the vast majority of developers they surveyed. The study reported that developers found flaky tests to have a wider impact on the reliability of the test suites. As part of their survey with developers, the authors also conducted a mini-multivocal review study to collect evidence from the literature on the challenges of dealing with flaky tests. However, this was a small, targeted review to address only the challenges of dealing with flaky tests. The study included a review of only a few (19) articles. A recent developers' survey (Habchi et al., 2022) echoed the results found in Eck et al. noting that flakiness can result from interactions between the system components, the testing infrastructure, and other external factors.

Ahmad et al. (2021) conducted a similar survey with developers aiming to understand developers' perception of test flakiness (e.g., how developers define flaky tests, and what factors are known to impact the presence of flaky tests). The study identified several key factors that are believed to be impacted by the presence of test flakiness, such as software product quality and the quality of the test suite.

The systematic review by Zolfaghari et al. (2020) identified what has been done so far on test flakiness in general and presented points for future research directions. The authors identified the main methods behind approaches for detecting flaky tests, methods for fixing flaky tests, empirical studies on test flakiness, root causes of flakiness and listed tools for detecting flaky tests. The study suggested an investigation into building a taxonomy of flaky tests that covers all dimensions (causes, impact, detection), formal modelling of flaky tests, setting standards for flakiness-free testing and investigating the application of AI-based approaches to the problem, and automated flaky test repair.

Zheng et al. (2021) also discussed current trends and research progress in flaky tests. The study analysed similar questions to the research reported in this paper on causes and detection techniques of flaky tests in 31 primary studies on flaky tests. Hence, this review was limited, and it did not discuss in detail the mechanism of current detection approaches or the wider impact of flaky tests on other techniques. There was a short scoping grey literature review by Barboni et al. (2021) that focused on investigating the definition of flaky tests in grey literature by analysing flaky-test-related blogs posted on *Medium*. The study is limited to understanding the definition of flaky tests (highlighting the problem of inconsistent terminology used in the surveyed articles), and covered a small subset of the posts (analysing only 12 articles in total).

Parry et al. (2021) conducted a more recent comprehensive survey of academic literature on the topic of test flakiness. The study addressed similar research questions to our review and those in the previous reviews by studying causes, detection and mitigation of flaky tests. The study reviewed 76 articles that focused on flaky tests.

The review presented in this paper covers a longer period of time than those previous reviews (Parry et al., 2021; Zolfaghari et al., 2020; Zheng et al., 2021), which includes work dating back further (on "non-deterministic tests"), before the term "flaky tests" became popular. The review contains a discussion of publications through the end of April 2022, whereas the most recent review of Parry et al. (2021) covers publications through April 2021. We found a significant number of academic articles published between the two reviews (229 articles published between 2021–2022). In general, our study complements previous work in that, (1) we gather more detailed evidence about causes of flaky tests and investigate the relationships between different causes, (2) we investigate both the impact of and responses to flaky tests in both research and practice, (3) we list the *indirect* impact of flaky tests on other analysis methods and techniques (e.g., software debugging and maintenance).

All previous reviews focused on academic literature. The review by Zolfaghari et al. (2020) covered a total of 43 articles, Parry et al. (2021) covered 76 articles, and Zheng et al. (2021) covered 31 articles. Our study complements these reviews by providing much wider coverage and an in-depth perspective on the topic of flaky tests. We include 602 academic articles and review 91 grey literature entries (details in Section 3.3). We cover not only studies that directly report on flaky tests, but also those that reference or discuss the issue of test flakiness while it is not the focus of the study. We also discuss a wide range of flaky test-related tools used in research and practice (including industrial and open-source tools), and discuss the impact of flaky tests from different perspectives.

A comparative summary of this review with the previous three reviews is shown in Table 1.

Table 1
Summary of prior reviews on test flakiness compared with our review.

Paper	Period covered	# of reviewed articles	Grey literature	Focus
Zolfaghari et al. (2020)	2013–2020	43	–	Causes and detection techniques
Zheng et al. (2021)	2014–2020	31	–	Causes, impact, detection and fixing approaches
Parry et al. (2021)	2009–4/2021	76	–	Causes, costs and consequences, detection and approaches for mitigation and repair
This review	1994–5/2022	109	91	Taxonomy of causes, detection and responses techniques, and impact on developers, process and product in research and practice

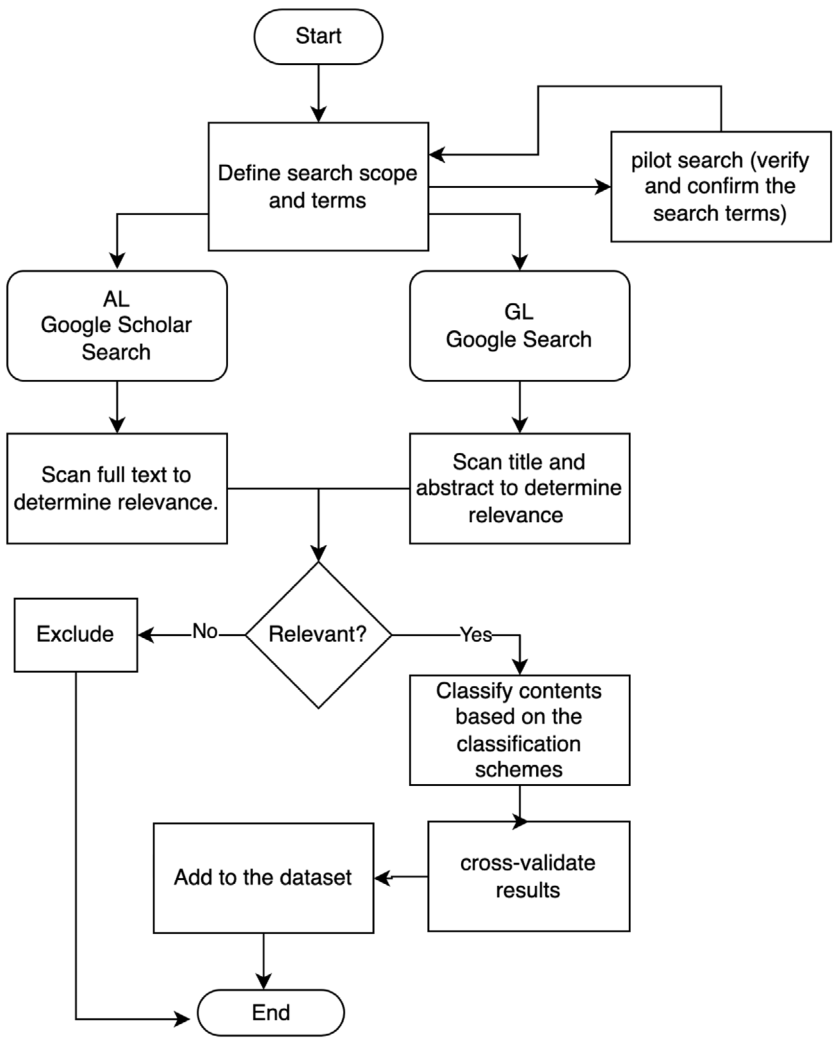

Fig. 1. An overview of our review process.

3. Study design

We designed this review following the Systematic Literature Review (SLR) guidelines by Kitchenham and Charters (2007), and the guidelines of Garousi et al. (2019) on multivocal review studies in software engineering. The review process is summarized in Fig. 1.

3.1. Research questions

This review addresses a set of research questions that we categorized along four main dimensions: *causes*, *detection*, *impact* and *responses*. We list our research questions below, with the rationale for each.

Causes of flaky tests:

RQ1. What are the common causes of flaky tests?

The rationale behind this question is to list the common causes of flaky tests and then group similar categories of causes together. We also investigate the cause–effect relationships between different flakiness causes as reported in the reviewed studies, as we believe that some causes are interrelated. For example, flakiness related to the User Interface (UI) could be attributed to the underlying environment (e.g., the Operating System (OS) used). Understanding the causes and their relationships is key to dealing with flaky tests (i.e., detection, quarantine or elimination).

Common causes of flaky tests:
RQ1. What are the common causes of flaky tests?

The rationale behind this question is to list the common causes of flaky tests and then group similar categories of causes together. We also investigate the cause–effect relationships between different flakiness causes as reported in the reviewed studies, as we believe some causes are interrelated. For example, flakiness related to the User Interface (UI) could be attributed to the underlying environment (e.g., the Operating System (OS) used). Understanding the causes and their relationships is key to dealing with flaky tests (i.e., detection, quarantine or elimination).

Detection of flaky tests
RQ2. How are flaky tests being detected?

To better understand flaky test detection, we divide this research question into the following two sub-questions.
RQ2.1. What *methods* have been used to detect flaky tests?
RQ2.2. What *tools* have been developed to detect flaky tests?

In RQs 2.1 and 2.2, we gather evidence regarding methods/tools proposed/used to detect flaky tests. We seek to understand how these methods work. We later discuss the potential advantages and limitations of current approaches.

Impact of flaky tests
RQ3. What is the impact of flaky tests?

As reported in previous studies, flaky tests are generally perceived to have a negative impact on software products, and processes (Fowler, 2011b; Micco, 2016; Harman and O'Hearn, 2018; Eck et al., 2019b). However, it is important to understand the extent of this impact and what exactly is affected (e.g., process, product, personnel).

Responses to flaky tests
RQ4. How do developers/organizations respond to flaky tests when detected?

Here we look at the responses and mitigation strategies employed by developers, development teams and organizations. We note that there are both technical (i.e., how to fix the test or the underlying code that causes the flakiness) and management (i.e., what are the processes followed to manage flaky test suites) responses.

3.2. Review process

Since this is a multivocal review where we search for academic and grey literature in different forums, the search process for each of the two parts (academic and grey literature) is different and requires different steps. The systematic literature review search targets peer-reviewed publications that have been published in relevant international journals, conference proceedings and theses in the areas of software engineering, software testing and software maintenance. The search also covers preprint and postprint articles available in open access repositories such as *arXiv*. For the grey literature review, we searched for top-ranked online posts on flaky tests. This includes blog posts, technical reports, white papers, and official tools documentation.

We used *Google Scholar* to search for academic literature and *Google* search engine to search for grey literature. Both *Google Scholar* and *Google search* have been used in similar multivocal studies in software engineering (Garousi and Küçük, 2018; Myrbakken and Colomo-Palacios, 2017; Garousi and Mäntylä, 2016) and other areas of computer science (Islam et al., 2019; Pereira-Vale et al., 2021). Google Scholar indexes most major publication databases relevant to computer science and software engineering (Neuhaus et al., 2006), particularly the ACM digital library, IEEE Xplore, ScienceDirect and SpringerLink, thus providing a much wider coverage compared to those individual libraries. A recent study on the use of Google Scholar in software engineering reviews found it to be very effective, as it was able to retrieve ∼96% of primary studies in other review studies (Yasin et al., 2020). Similarly, it has been suggested that a regular *Google Search* is sufficient to search for grey literature material online (Mahood et al., 2014; Adams et al., 2016).

3.2.1. Searching academic publications

We closely followed Kitchenham and Charters's guidelines (Kitchenham and Charters, 2007) to conduct a full systematic literature review. The goal here is to identify and analyse primary studies relating to test flakiness. We defined a search strategy and search string that covers the terminology associated with flaky tests. The search string was tested and refined multiple times during our pilot runs to ensure coverage. We then defined a set of inclusion and exclusion criteria. We included a quality assessment of the selection process to ensure we covered all relevant primary studies. We explain those steps in detail below.

We define the following criteria for our search:

> **Search engine:** Google Scholar.
> **Search String:** "flaky test" OR "test flakiness" OR "flaky tests" OR "nondeterministic tests" OR "non deterministic tests" OR "nondeterministic test" OR "non deterministic test"
> **Search scope:** all academic articles published until 30 April 2022.

In case an article appears in multiple venues (e.g., a conference paper that was also published on arXiv under a different title, or material from a thesis that was subsequently published in a journal or conference proceedings), we only include the published articles/papers over the other available versions. This was to ensure that we included as much peer-reviewed material as possible. We conducted this search over two iterations. The first iteration covers the period until 31 December 2020, whereas the second iteration covers the period between 1 January 2021 and 30 April 2022. Results from the two searches were then combined.

Our review for academic articles follow the following steps:

1. Search and retrieve relevant articles using the defined search terms using Google Scholar.
2. Read the title, abstract and the full text (if needed) to determine relevance by one of the co-authors and apply inclusion and exclusion criteria.
3. Cross-validate a randomly selected set of articles by another co-author.
4. Apply inclusion and exclusion criteria.
5. Classify all included articles based on the classification we have for each question (details of those classifications are provided for each research question in Section 4).

3.2.2. Searching for grey literature

Here we followed the recommendations made in previous multivocal review studies (Garousi and Küçük, 2018; Garousi et al., 2019) and reported the results obtained only from the first 10 pages (10 items each) of the Google search. It was reported that relevant results usually only appear in the first few pages of the search (Garousi et al., 2019). We observed that the results in the last five pages were less relevant compared to those that appeared in the first five.

For the grey literature search, we define the following criteria:

> **Search engine:** Google Search.
> **Search String:** *"flaky test" OR "test flakiness" OR "non-deterministic test".*

Search scope: pages that appear in the first 10 pages of the search. Note that this search was conducted over two iterations. We searched for material published up until 31 December 2020, and then in the second iteration we searched for material published up to 30 April 2022 (we removed duplication found between the two searches).

The grey literature review consists of the following steps:

1. Search and retrieve relevant results using the search terms in Google Search.
2. Read the title and full article (if needed) to determine relevance.
3. Cross-validate a randomly selected set of articles by another co-author.
4. Check external links and other external resources that have been referred to in the articles/posts. Add new results to the dataset.
5. Classify all posts based on the classification scheme.

3.2.3. Selection criteria

We selected articles based on the three following inclusion criteria:

- Studies discussed test flakiness as the main topic of the article.
- Studies discussed test flakiness as an impact of using a specific technique, tool or experiment.
- Studies discussed test flakiness as a limitation of a technique, tool or experiment.

We apply the following exclusion criteria:

– Articles only mention flakiness, but without substantial discussion on the subject.
– Articles that are not accessible electronically, or the full text is not available for downloading.[2]
– Studies on nondeterminism in hardware and embedded systems.
– Studies on nondeterminism in algorithms testing (e.g., when nondeterminism is intentionally introduced).
– Duplicate studies (e.g., reports of the same study published in different places or on different dates, or studies that appeared in both academic and grey literature searches).
– Secondary studies on test flakiness (previous review articles).
– Editorials, prefaces, books, news, tutorials and summaries of workshops and symposia.
– Multimedia material (videos, podcasts, etc.) and patents.
– Studies written in a language other than English.

For the grey literature study, we also exclude the following:

– Academic articles, as those are covered by our academic literature search using Google Scholar.
– Tools description pages (such as GitHub pages) with little or no discussion about the causes of flakiness or its impact.
– Web pages that mention flaky tests with no substantial discussion (e.g., just provide a definition of flakiness).

[2] In case the article is not available either through a known digital library such as ACM Digital Library, IEEE Xplore, ScienceDirect and SpringerLink; or not publicly available through other open-access repositories such as arXiv or ResearchGate.

3.2.4. Pilot run

Before we finalized our search keywords and search strings, and defined our inclusion and exclusion criteria, we conducted a pilot run using a simplified search string to validate the study selection criteria, refine/confirm the search strategy and refine the classification scheme before conducting the full-scale review. Our pilot run was conducted using a short, intentionally inclusive string (*"flaky test" OR "test flakiness" OR "non-deterministic test"*) using both Google and Google Scholar. These keywords were drawn from two key influential articles and blog posts (either used in the title or as keywords) that the authors are aware of - i.e., the highly cited work on the topic by Luo et al. (2022), and the well-known blog post by Martin Fowler (Fowler, 2011b)). This was done for the period until 31 December 2020.

In the first iteration, we retrieved 10 academic articles and 10 grey literature results, and then in the second iteration, we obtained another 10 academic articles (next 10 results) and 10 grey literature results. We validated the results of this pilot run based on the articles' relevance as well as our familiarity with the field. We validated the retrieved articles and classified all retrieved results using the defined classification scheme in order to answer the four research questions. We used this pilot run to improve and update our research questions and our classification scheme. We classified a total of 20 articles in each group (academic and grey literature). With respect to the former, we were able to identify 14 of the 20 articles found by the search as being familiar to us, lending a degree of confidence that our search would at minimum find papers relevant to our research questions.

3.3. Data extraction and classification

We extracted relevant data from all reviewed articles in order to answer the different research questions. Our search results (following the different steps explained above) are shown in Fig. 2.

As explained in Section 3.2, we conducted our search over two iterations, covering two periods. The first period covers articles published up until 31 December 2020, while the second cover the period from 1 January 2021 until 30 April 2022. In the first iteration, we retrieved a total of 1090 results, with 990 articles obtained from the Google Scholar search and 100 grey literature articles obtained from Google Search (i.e., the first 10 pages). After filtering the relevant articles and applying the inclusion and exclusion criteria, we ended up with a total of 408 articles to analyse (354 academic articles and 54 grey literature articles). In the second iteration (from January 2021 until April 2022), we retrieved 330 academic articles from Google Scholar and 100 articles from Google Search (results from the first 10 pages). We removed the duplicates (e.g., results that might appear twice such as the same publication appearing in multiple publication venues, or a grey literature article that appeared in the top 10 pages over the two iterations). For this iteration, after filtering the relevant articles, we ended up with 243 results (206 academic articles and 37 grey literature posts). Collectively, we identified a total of 560 academic and 91 grey literature articles for our analysis.

The analysis of articles was done by three coders (co-authors). We split the articles between coders, where each coder read the articles and obtained data to answer our research questions. For each article, we first tried to understand the context of flakiness. We then looked for the following: (1) the discussed causes of flakiness (RQ1), (2) how flakiness is detected - in terms of the *methods* used (e.g., static vs. dynamic) or the tools implemented to detect flakiness (RQ2), (3) the noted impact of flakiness (R3), and (4) the approach used to respond to or deal with flaky tests (RQ4).

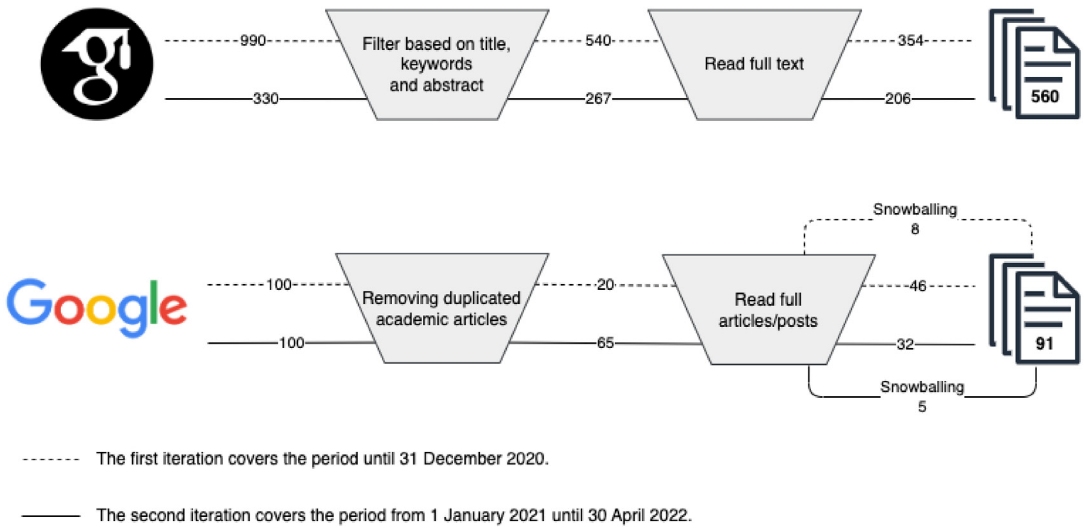

Fig. 2. Results of the review process.

3.4. Reliability assessment

We conducted a reliability check of our filtration and classification. We cross-validated a randomly selected sample of 88/990 academic articles and 49/100 grey literature articles, as obtained from the first iterations (obtaining a confidence level = 95% and confidence interval = 10). Two of this paper's authors cross-validated those articles, with each classifying half of these articles (44 of academic literature and 25 of the grey literature articles). In addition, a third co-author of this paper (who was not involved in the initial classification) cross-validated another 25 randomly selected academic/grey articles. On those cross-validations, we reached an agreement level of ≥ 90%.

We provide the full list of articles that we reviewed (both academic and grey) online https://docs.google.com/spreadsheets/d/1eQhEAUSMXzeiMatw-Sc8dqvftzLp8crC3-B9v5qHEuE.

4. Results

4.1. Overview of the publications

We first provide an overview of the timeline of publications on flaky tests in order to understand how the topic has been viewed and developed over the years. The timeline of publications is shown in Fig. 3. Based on our search for academic articles, we found that there have been articles that discuss the issue of *nondeterminism* in test outcomes dating back to 1994, with 34 articles found between 1994 and 2014. However, the number of articles has significantly increased since 2014. There has been an exponential growth in publications addressing flaky tests in the past 6 years (between 2016 and 2022). We attributed this increase to the rising attention to flaky tests by the research community since the publication of the first empirical study on the causes of flaky tests in Apache projects by Luo et al. in 2014 (Luo et al., 2014), which was the first study that directly addressed the issue of flaky tests in great detail (in terms of common causes and fixes). Over 93% of the articles were published after the publication of this study, with 41% of those articles (229) published between January 2021 and April 2022 only, indicating an increased popularity over the years.

In terms of publication types and venues, more than 40% of these publications have been published in reputable and highly rated software engineering venues. Top publications venues include the premier software engineering conferences: International Conference on Software Engineering (ICSE), Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) and the International Conference on Automated Software Engineering (ASE). Publications on flaky tests has also appeared in the main software testing conferences International Symposium on Software Testing and Analysis (ISSTA) and International Conference on Software Testing, Verification and Validation (ICST). They also appear in software maintenance conferences, International Conference on Software Maintenance and Evolution (ICSME) and International Conference on Software Analysis, Evolution and Reengineering (SANER). A few articles (∼10%) were published in premier software engineering journals, including Empirical Software Engineering (EMSE), Journal of Systems and Software (JSS), IEEE Transaction in Software Engineering (TSE) and Software Testing, Verification and Reliability (STVR) journal. The distribution of publications in key software engineering venues is shown in Fig. 4).

Table 2
Flaky tests in terms of languages studied.

Language	# Articles
Java	212
Python	25
JavaScript	11
.NET languages	6
Other languages	27
Multiple languages	58
Not specified/unknown	221

To expand on the methodology for including and excluding articles, we did not perform a quality assessment of articles based on venues or citation statistics. We focused on primary studies (excluding the three reviews and secondary studies). Furthermore, looking deeper into focused studies provided insights into their quality.

In terms of programming languages, the vast majority of the studies have focused on Java (49% of those studies), with only fewer other studies that discuss flakiness in other languages such as Python, JavaScript and .NET, with 49 (14%) studies used multiple languages (results listed in Table 2).

We classified all articles into three main categories: (1) studies focusing on test flakiness, where flakiness is the focal point (e.g., an empirical investigation into test flakiness due to a particular cause such as concurrency or order dependency), (2) studies that explain how test flakiness impacts tools/techniques (e.g., the

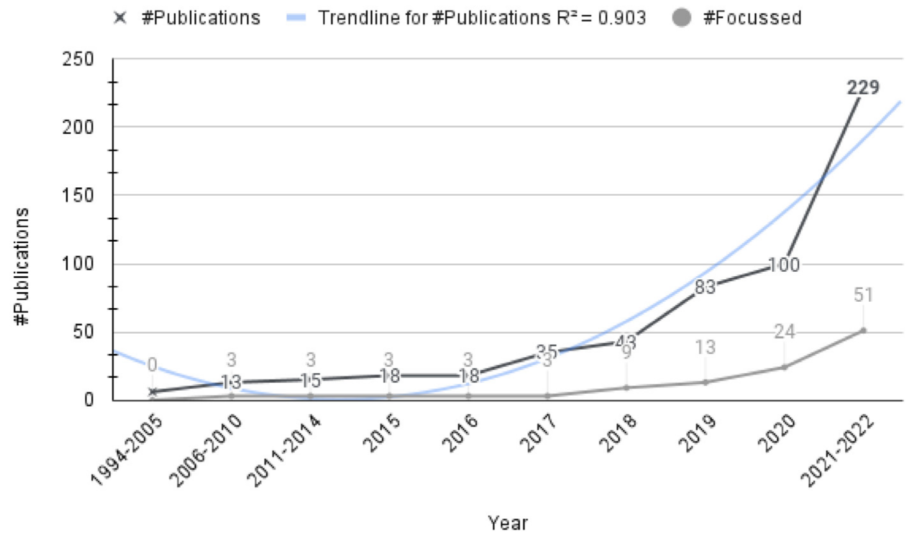

Fig. 3. Timeline of articles published on flaky tests, including the focused articles (i.e., test flakiness is the main subject of the study). * The 2021–2022 numbers include article published between Jan 2021 and April 2022.

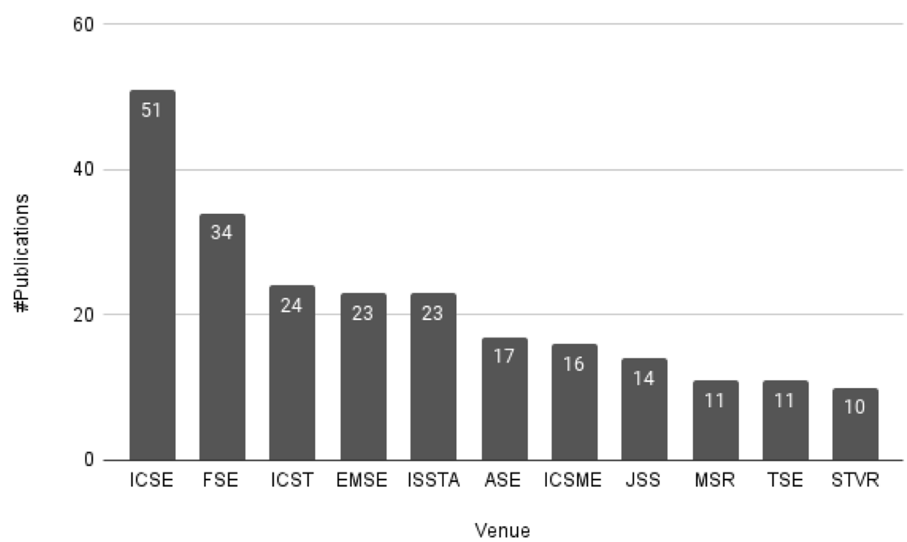

Fig. 4. Distribution of publications based on publication venues.

impact of test flakiness on program repair or fault localization) or (3) studies that just mention or reference test flakiness (but it is not a focus of the study). A breakdown of the focus of the articles and the nature of the discussion on test flakiness in academic literature is shown in Table 3.

We observed that only 109 articles (∼20%) from all the collected studies focused on test flakiness as the main subject of the study. The majority of the reviewed articles (297, representing ∼53%) just mentioned test flakiness as a related issue or as a threat to validity. The remaining 154 articles(∼27%) discussed flakiness in terms of their impact on a proposed technique or tool that is the subject of the study.

As for the grey literature results, all articles we found were published following the publication of Martin Fowler's influential blog post on test nondeterminism in early 2011 (Fowler, 2011b).

Similar to the recent increased attention on those academic literature articles, we found that almost 50% of the grey literature articles (26) have been published between 2019 and 2020, indicating a growing interest in flaky tests, and shedding light on the importance of the topic in practice.

In the following subsections, we present the results of the study by answering each of our four research questions in detail.

Table 3
Focus of academic articles.

Type	Description	# of articles
Focused	Studies focusing on test flakiness	109
Impact	Studies that explain how test flakiness impacts tools/techniques	154
Referenced	Studies that just mention or reference test flakiness	297

4.2. Causes of test flakiness (RQ1)

We analysed flaky test causes, as noted in both academic and grey literature articles. We looked for the quoted reasons for why a test is flaky, and in most cases, we note multiple (sometimes connected) causes being the reason for flakiness. We then grouped those causes into categories based on their overall nature.

Table 4 lists the number of studies (in the focused category) that cite the cause. The most common causes of flakiness is related to *order dependency*, followed by *Async/Wait*, *Randomness*

Table 4
Count of flaky tests causes as in the *focused* articles.

Cause	Count
Order dependency	16
Async/Wait	15
Randomness	10
Environment	10
Concurrency	8
Platform	7
Network	6

(mostly in ML applications), *environment*, *concurrency*, *platform* and *network*.

The most frequently discussed causes, as shown in Table 4, are also covered in the first empirical study on this subject by Luo et al. (2014). Luo et al.'s study provides a classification of causes from analysing 201 commits that fix flaky tests across Apache projects, which are diverse across languages and maturity. Their methodology is centred around examining commits that *fix* flaky tests. In addition to classifying the root causes of test flakiness into categories, they present approaches to manifest flakiness and strategies to fix flaky tests. The classification consists of 10 categories that are the *root causes* of flaky tests in the commits, which are *async-wait*, *concurrency*, and *test order dependency*, *resource leak*, *network*, *time*, *IO*, *randomness*, *floating point operations* and *unordered collections*. Thorve et al. (2018) listed additional causes identified from a study of Android commits fixing flaky tests: *dependency*, *program logic*, and *UI*. Dutta et al. (2020) noted subcategories of *randomness* and Eck et al. (2019a) identified three additional causes: *timeout*, *platform dependency* and *too restrictive range* from a survey of developers.

We mapped all causes found in all surveyed publications and categorized them into the following major categories (based on their nature): *concurrency*, *test order dependency*, *network*, *randomness*, *platform dependency*, *external state/behaviour dependency*, *hardware*, *time* and *other*. A summary of the causes we classified is provided in Table 5 and discussed below.

Concurrency. This categorizes flakiness due to concurrency issues resulting from concurrency-related bugs. These bugs can be race conditions, data races, atomicity violations or deadlocks. ***Async-wait*** is investigated as one of the major causes of flakiness under concurrency. This occurs when an application or test makes an asynchronous call and does not correctly wait for the result before proceeding. This category accounts for nearly half of the studied flaky test fixing commits (Luo et al., 2014). Thorve et al. (2018) and Luo et al. (2014) classified async-wait related flakiness under concurrency. Lam et al. (2020a, 2019a) reported async-wait as the main cause of flakiness in Microsoft projects. Other articles cited async-wait in relation to root-cause identification (Lam et al., 2019a), detection (Lam et al., 2020d) and analysis (Malm et al., 2020). Luo et al. (2014) identified an additional subcategory *"bug in condition"* for concurrency-related flakiness, where the guard for code that determines which thread can call it is either too restrictive or permissive. Concurrency is also identified as a cause in articles on detection (Lam et al., 2019a; Lam et al., 2020d; Dutta et al., 2020 and Thorve et al., 2018). Another subcategory identified from browser applications is event races (Dong et al., 2021b).

Test order dependency. The test independence assumption implies that tests can be executed in any order and produce expected outcomes. This is not the case in practice (Lam et al., 2020b) as tests may exhibit different behaviour when executed in different orders. This is due to a shared state where it can either be in memory or external (e.g. file system, database). Tests can expect a particular state before they can exhibit the expected outcome, which can be different if the state is not set up correctly or reset. There can be multiple sources of test order dependency. Instances of shared state can be in the form of explicit or implicit data dependencies in tests or even bugs such as resource leaks or failure to clean up resources between tests. Luo et al. (2014) listed these as separate root causes: resource leaks and I/O. ***Resource leak*** can be a source of test order dependency when the code under test (CUT) or test code is not properly managing shared resources (e.g. obtaining a resource and not releasing it). Empirical studies that discuss this resource leak-related flakiness include (Luo et al., 2014) and Lam et al. (2020a), as well other studies on root cause analysis, such as Lam et al. (2019a) and Strandberg et al. (2020), that find instances of flakiness in test code due to improper management of resources. Luo et al. (2014) identified I/O as a potential cause of flakiness. An example is code that opens and reads from a file and does not close it, leaving it to garbage collection to manage it. If a test attempts to open the same file, it would only succeed if the garbage collector had processed the previous instance. In Luo et al. (2014), 12% of test flakiness in their study is due to order dependency. Articles that cite order dependency include those that propose detection methods (Gambi et al., 2018a; Lam et al., 2019b), and one that is an experimental study on flakiness in generated test suites (Paydar and Azamnouri, 2019). A shared state can also arise due to *incorrect/flaky API usage* in tests. Tests may intermittently fail if programmers use such APIs in tests without accounting for such behaviour. Dutta et al. (2020) discussed this in the study of machine learning applications and cite an example where the underlying cause is the shared state between two tests that use the same API and one of the tests not resetting the fixture before executing the second.

Network. Another common cause for flaky tests relates to network issues (connections, availability, and bandwidth). This has two subcategories: local and remote issues. Local issues pertain to managing resources such as sockets (e.g. contention with other programs for ports that are hard-coded in tests) and remote issues concern failures in connecting to remote resources. Morán Barbón et al. (2020) studied network bandwidth in localizing flakiness causes. In a study consisting of Android projects (Thorve et al., 2018), network is identified as a cause of flakiness of 8% of the studied flaky tests.

Randomness. Tests or the code under test may depend on randomness, which can result in flakiness if the test does not consider all possible random values that can be generated. This is listed as a main cause by Luo et al. (2014). Dutta et al. (2020) identified subcategories of randomness in their investigation of flaky tests in probabilistic and machine learning applications. Such applications rely on machine learning frameworks that provide operations for inference and training, which are largely nondeterministic in nature. Writing tests can be challenging for such applications which use these frameworks. The applications are written in Python, and they study applications that use the main ML frameworks for the language. They analysed 75 bugs/commits that are linked to flaky tests and obtained three cause subcategories: (1) *algorithmic nondeterminism*, (2) *incorrect/flaky API usage* and (3) *hardware*. They state that these categories are subcategories of the *randomness* category in Luo et al. (2014). The most common cause identified was algorithmic nondeterminism. They also present a technique to detect flaky tests due to such assertions. They evaluate the technique on 20 projects and found 11 previously unknown flaky tests. The subcategories identified are *Algorithmic non-determinism* and *Unsynchronized seeds* in ML applications. In these applications, developers use small datasets and models as test inputs, expecting the results to converge to values within an expected range. Assertions are added to check if the inferred values are close to the expected ones. As there is a chance that the computed value may fall outside the expected range, this may result in flaky outcomes. Tests in ML applications may also use multiple libraries that need sources of randomness, and flakiness can arise if different random number seeds are used across these modules or if the seeds are not set. We include a related category here, *too restrictive ranges*, identified in Eck et al. (2019a). This is due to output values falling outside ranges or values in assertions determined at design time.

Platform dependency. This causes flakiness when a test is designed to pass on a specific platform but when executed on another platform it unexpectedly fails. A platform could be the hardware and OS and also any component on the software stack that test execution/compilation depends on. Tests that rely on platform dependency may fail on alternative platforms due to missing preconditions or even performance issues across them. The cause was initially described in Luo et al. (2014), though it was not in the list of 10 main causes as it was a small category. It is discussed in more detail in Eck et al. (2019a). In Thorve et al. (2018), it was reported that dependency flakiness for Android projects are due to hardware, OS version or third-party libraries. The study consisted of 29 Android projects containing 77 flakiness-related commits. We also include *implementation dependency*, differences in compilation (Gruber and Fraser, 2022) and infrastructure flakiness (Gruber et al., 2021a) under this category. Infrastructure flakiness could also be due to issues in setting up the required infrastructure for test execution, which could include setting up Virtual Machines (VM)/containers and downloading dependencies, which can result in flakiness. Environmental dependency flakiness due to dynamic aspects (performance or resources) is also included in this category.

Dependencies on external state/behaviour. We include flakiness due to changes in external dependencies like state (e.g. reliance on external data from databases or obtained via REST API's) or behaviour (changes or assumptions about the behaviour of third-party libraries) in this category. Thorve et al. (2018) included this under platform dependency.

Hardware. Some ML applications/libraries may use specialized hardware, as discussed in Dutta et al. (2020). If the hardware produces nondeterministic results, this can cause flakiness. An example is where an accelerator is used that performs floating-point computations in parallel. The ordering of the computations can produce nondeterministic values, leading to flakiness when tests are involved. Note that this is distinct from platform dependency, which can also be at the hardware level, for instance, different processors or Android hardware.

Time. Variations in time are also a cause of test flakiness (e.g. midnight changes in the UTC time zone, daylight saving time, etc.), and due to differences in precision across different platforms. Time is listed as a cause in root cause identification by Lam et al. (2019a). New subcategories, *timeouts*, are listed by developers in the survey done in Eck et al. (2019a). Time precision across OS, platforms and different time zones are listed under this category (Berglund and Vateman, 2020). Another cause related to time is that test cases may time out nondeterministically e.g. failing to obtain prerequisites or execution not completing within the specified time due to flaky performance. A similar cause is test suite timeouts where no specific test case is responsible for it. Both of these causes were identified in the developers' survey reported in Eck et al. (2019a).

Other causes. We include causes listed in articles, which may already have relationships with the major causal categories listed above. Thorve et al. (2018) listed *program logic* as one of them. This category consists of cases where programmers have made incorrect assumptions about the code's behaviour, which results in cases where tests may exhibit flakiness. The authors cited an example where the Code Under Test (CUT) may nondeterministically raise an I/O exception and the exception handling throws a runtime exception, causing the test to crash in that scenario. *UI flakiness* can be caused due to developers either not understanding UI behaviour or incorrectly coding UI interactions (Thorve et al., 2018). They can also be caused by concurrency (e.g., event races or async-wait) or platform dependency (e.g., dependence on the availability of a display, dependence on a particular browser Morán Barbón et al., 2020). *Floating-point operations.* floating-point operations can cause flakiness as they can be non-deterministic due to non-associative addition, overflows and underflows as described in Luo et al. (2014). It is also discussed in the context of machine learning applications (Dutta et al., 2020). Concurrency, hardware and platform dependency can be a source of nondeterminism in floating-point operations. Luo et al. (2014) identified *unordered collections*, where there are variations in outcomes due to a test's incorrect assumptions about an API. An example of this is the sets which can have specifications that are underdetermined. Code may assume behaviour such as the order of the collection from a certain execution/implementation, which is not deterministic.

4.2.1. Ontology of causes of flaky tests

Fig. 5 illustrates the different causes of flakiness. The figure uses Web Ontology Language (OWL) (McGuinness et al., 2004) terminology such as classes, subclasses and relations. We identify classes for causes of flakiness and flaky tests. Subclass relationships between classes are named 'kindOf' and 'causes' is the relation for denoting causal relationships.

Note that not all identified causes are shown in the diagram. For instance, causes listed under the other category may be due to sources already shown in the diagram. For instance, UI flakiness can be due to platform or environmental dependency. An example that demonstrates the complex causal nature of flakiness is in Dutta et al. (2020), where the cause of flakiness is due to a hardware accelerator for deep learning, which performed fast parallel floating-point computations. Different orderings of floating point operations can result in different outputs, which leads to test flakiness. In this case, the causes are a combination of *hardware*, *concurrency*, and *floating point operations*. Network uncertainty can be attributed to multiple reasons, for instance, connection failure and bandwidth variance. Stochastic algorithms exhibit randomness, and concurrency-related flakiness can be due to concurrency bugs such as races and deadlocks. Finally, order dependency is due to improper management of resources (e.g. leaks and not cleaning up after I/O operations) or hidden state sharing that may manifest in flakiness.

There are a number of factors that vary that underlie those causes. For instance, *random seed variability* can cause flakiness related to randomness and scheduling variability causes concurrency-related flakiness. *Test execution order variability*, which causes order-dependent test flakiness and types of *platform variability* (e.g. hardware and browser that can, for instance, manifest in UI flakiness) are additional dimensions of variability.

RQ1 summary. Numerous causes of flakiness have been identified in the literature. Factors related to concurrency, test order dependency, network availability and randomness are the most common causes of flaky test behaviour. Other factors related to specific types of systems, such as *algorithmic nondeterminism* and *unsynchronized seeds*, impact testing in ML applications. There is also a casual relationship between some of these factors (i.e., they impact each other — for example, UI flakiness is mostly due to concurrency issues).

Table 5
Causes of flaky tests.

Main category	Sub-category	Description	Example articles
Concurrency	Synchronization	Asynchronous call in test (or CUT) without proper synchronization before proceeding	Luo et al. (2014), Thorve et al. (2018), Lam et al. (2020a) and Lam et al. (2019a) Lam et al. (2020d) and Malm et al. (2020)
	Event races	Event racing due to a single UI thread and async events triggering UI changes	Dong et al. (2021a) and Endo and Møller (2020)
	Bugs	Other concurrency bugs (deadlocks, atomicity violations, different threads interacting in a non-desirable manner.)	Luo et al. (2014)
	Bug in condition	A condition that inaccurately guards what thread can execute the guarded code.	Luo et al. (2014)
Test order dependency	Shared state	Tests having the same data dependencies can affect test outcomes.	Gambi et al. (2018a), Lam et al. (2019b) and Paydar and Azamnouri (2019)
	Shared state	Local files	Luo et al. (2014) and Person and Elbaum (2015)
	Resource leaks	When an application does not properly manage the resources it acquires	Luo et al. (2014), Lam et al. (2020a, 2019a) and Strandberg et al. (2020)
Network	Remote	Connection failure to remote host (latency, unavailability)	Luo et al. (2014) and Person and Elbaum (2015)
	Local	Bandwidth, local resource management issues (e.g. port collisions)	
Randomness	Data	Input data or output from the CUT	Lam et al. (2019a) and Eddins (2009)
	Randomness seed	If the seed is not fixed in either the CUT or test it may cause flakiness.	Dutta et al. (2020)
	Stochastic algorithms	Probabilistic algorithms where the result is not always the same.	Scott and Luke (2019)
	Too restrictive range	Valid output from the CUT are outside the assertion range.	Eck et al. (2019a)
Platform dependency	Hardware	Environment that the test executes in (Development/Test/CI or Production.)	Luo et al. (2014), Dutta et al. (2020), Mårtensson et al. (2016) and Strandberg et al. (2020)
	OS	Varying operating system	Morán Barbón et al. (2019, 2020)
	Compiler	Difference in compiled code	Gruber and Fraser (2022)
	Runtime	e.g., Languages with virtual runtimes (Java, C# .. etc.)	Luo et al. (2014)
	CI infra flakiness	Build failures due to infrastructure flakiness.	Gruber et al. (2021a)
	Browser	A browser may render objects differently affecting tests.	Morán Barbón et al. (2019)
External state/behaviour dependency	Reliance on production service	Tests rely on production data that can change.	
	Reliance on external resources	Databases, web, shared memory... etc	Mascheroni and Irrazábal (2018) and Berglund and Vateman (2020)
	API changes	Evolving REST APIs due to changing requirements	
	External resources	Relying on data from external resources (e.g., REST APIs, databases)	Mascheroni and Irrazábal (2018) and Mendes (2019)
Environmental dependencies		Memory and performance	Morán Barbón et al. (2019)
Hardware	Screen resolution	UI elements may render differently on different screen resolutions causing problems for UI tests	
	Hardware faults		Strandberg et al. (2020)
Time	Timeouts	Test case/test suite timeouts.	Eck et al. (2019a)
	System date/time	Relying on system time can result in non-deterministic failure (e.g. time precision and changing UTC time)	Lam et al. (2019a) and Berglund and Vateman (2020)
Other	Floating point operations	Use of floating point operations can result in non-deterministic cases	Dutta et al. (2020) and Luo et al. (2014)
	UI	Incorrectly coding UI interactions	Thorve et al. (2018)

(*continued on next page*)

Table 5 (*continued*).

	Program logic	Incorrect assumptions about APIs	Thorve et al. (2018)
	Tests with manual steps		Habchi et al. (2021)
	Code transformations	Random amplification/instrumentation can cause flaky tests	Ivanković et al. (2019)

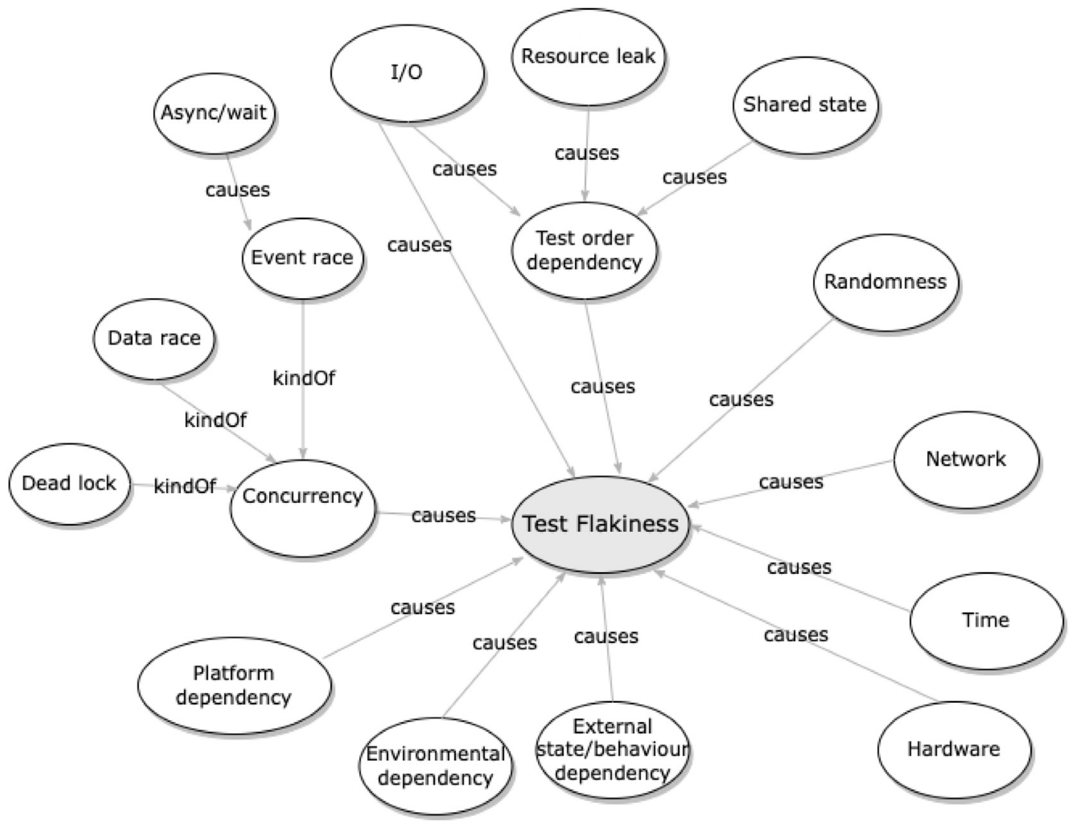

Fig. 5. Relationships between the different causes of flaky tests.

4.3. Flaky tests detection (RQ2)

One of the dimensions we studied is how flaky tests are identified and/or detected. In this section, we present methods used to detect and identify or locate causes of flakiness. We make a distinction between these three goals in our listing of techniques found in the reviewed literature. We look at methods identified in both academic and grey literature. RQ2 is divided into two sub-questions, we answer each subquestion separately, as shown below:

4.3.1. Methods used to detect flaky tests (RQ2.1)

There are two distinct approaches towards the detection of flaky tests, which are either by using dynamic techniques that involve the execution of tests or using static techniques that rely only on analysing the test code without actually executing tests. Fig. 6 depicts a broad overview of these strategies. Dynamic methods are based mostly on multiple test runs whilst also using techniques to perturb specific variability factors (e.g. environment, test execution order, event schedules or random seeds) that quickly manifest flakiness. There is one study on using program repair (Parry et al., 2020) to induce test flakiness and two studies on using differential coverage to detect flakiness without resorting to reruns (Bell et al., 2018). Under static approaches, studies have employed machine learning (3 studies), model checking for implementation-dependent tests and similarity patterns techniques for identifying flaky tests. There are only two studies that leverage hybrid approaches (one for order-dependent tests and another for async-wait).

Static methods: Static approaches that do not execute tests are mostly classification-based that use machine learning techniques (Ahmad et al., 2020; Verdecchia et al., 2021; King et al., 2018). Other static methods use pattern matching (Person and Elbaum, 2015) and association rule learning (Herzig and Nagappan, 2015). Model checking using Java PathFinder (Visser et al., 2003) has also been used for detecting flakiness due to implementation-dependent tests (Gyori et al., 2017).

Ahmad et al. (2020) evaluated a number of machine learning methods for predicting flaky tests. They used projects from the iDFlakies dataset (Lam et al., 2019b). There is also a suggestion that the evaluation also covered another language (Python) besides the data from the original dataset (which is in Java), though this is not made clear, and the set of Python programs or tests is not listed. The study was built on the work of Pinto et al. (2020), which evaluates five machine learning classifiers (Naive Bayes, Random Forest, Decision Tree, Support Vector Machine and Nearest Neighbour) that predict flaky tests. In comparison to Pinto et al. (2020), the study of Ahmad et al. (2020) answers two additional research questions: how classifiers perform with regard to another programming language, and the predictive power of the

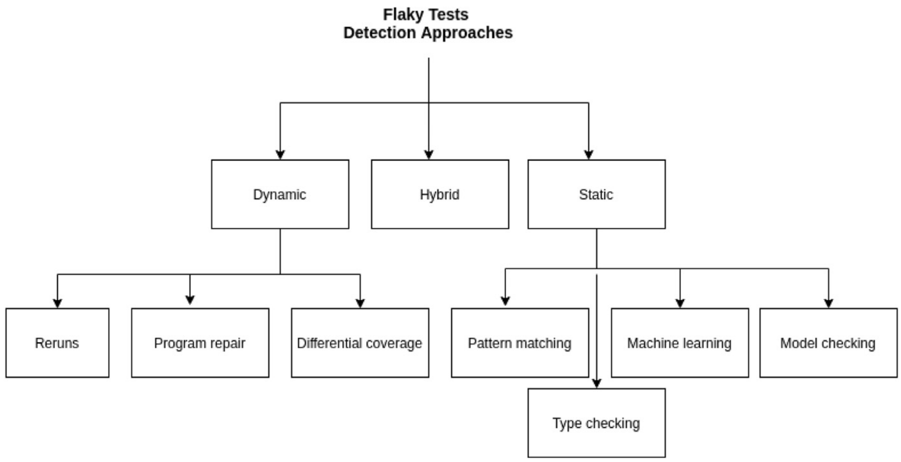

Fig. 6. Taxonomy of detection methods.

classifiers. Another static technique based on patterns in code has been used to predict flakiness (Person and Elbaum, 2015).

Dynamic methods: Dynamic techniques to detect flakiness are built on executions of tests (single or multiple runs). Those techniques are centred around making reruns less expensive by accelerating ways to manifest flakiness, i.e., fewer reruns or rerunning fewer tests. Methods to manifest flakiness include varying causal factors such as random number seeds (Dutta et al., 2020), event order (Endo and Møller, 2020), environment (e.g. browser, display) (Morán Barbón et al., 2020), and test ordering (Gambi et al., 2018a; Lam et al., 2019b). Test code has also been varied using program repair (Parry et al., 2020) to induce flakiness. Fewer tests are run by selecting them based on differential code coverage or those with state dependencies.

Hybrid methods: Dynamic and static techniques are known to make different trade-offs between desirable attributes such as recall, precision and scalability (Ernst, 2003). As in other applications of program analysis, hybrid techniques have been proposed to combine the strength of static and dynamic techniques whilst reducing their weaknesses. One of the tools, FLAST (Verdecchia et al., 2021), proposes a hybrid approach where the tool uses a static technique but suggests that dynamic analysis can be used to detect cases missed by the tool. Malm et al. (2020) proposed a hybrid analysis approach to detect delays used in tests that cause flakiness. Zhang et al. (2014) proposed a tool for dependent test detection, and they use static analysis to determine side-effect-free methods, whose field accesses are ignored when determining inter-test dependence in the dynamic analysis. Some tools, stated earlier under static methods (e.g., Ahmad et al. (2020)), may need access to historic execution data for analysis or training.

4.3.2. Tools to detect flaky tests (RQ2.2)

Table 6 lists the tools that detect test flakiness as described in the academic literature. Most of the tools detect flakiness manifested in test outcomes. Most of the tools found in academic articles work on Java programs, with only three for Python and a single tool for JavaScript. These tools can be grouped by the source of flakiness they target: UI, test order, concurrency and platform dependency (implementation dependency to a particular runtime). Some of these tools identify the cause of flakiness as well (which may already be a part of the tool's output if the source of flakiness they detect is closely associated with a cause: e.g., test execution order dependency arising from a shared state can be detected by executing tests under different orders).

FlakyLoc (Morán Barbón et al., 2020) does not detect flaky tests but identifies causes for a given flaky test. The tool executes the known flaky test in different environment configurations. These configurations are composed of environment factors (i.e., memory sizes, CPU cores, browsers and screen resolutions) that are varied in each execution. The results are analysed using a spectrum-based localization technique (Wong et al., 2016), which analyses the factors that cause flakiness and assigns a ranking and a suspiciousness value to determine the most likely factors. The tool was evaluated on a single flaky test from a Java web application (with several end-to-end flaky tests). The results for this particular test indicate that the technique can successfully rank the cause of flakiness (e.g., low screen resolution) for the test.

RootFinder (Lam et al., 2019a) identifies causes as well as the location in code that cause flakiness. It can identify nine causes (network, time, I/O, randomness, floating-point operations, test order dependency, unordered collections, and concurrency). The tool adds instrumentation at API calls during test execution, which can log interesting values (time, context, return value) and add additional behaviour (e.g., delay to identify causes involving concurrency and async wait). Post-execution, the logs are analysed by evaluating predicates (e.g., if the return value was the same at this point compared to previous times) at each point where it was logged. Predicates that evaluate consistent values in passing and failing runs are likely useful in identifying the causes, as they can explain what was different during passing and failing runs.

DeFlaker (Bell et al., 2018) detects flaky tests using differential coverage to avoid reruns (as reruns can be expensive). If a test has a different outcome compared to a previous run and the code covered by the test has not changed, then it can be determined to be flaky. The study also examines if a particular rerun strategy has an impact on flakiness detection. With Java projects, there can be many such strategies (e.g., five reruns in the same JVM, forking with each run in a new JVM, rebooting the machine and cleaning files generated by builds between runs).

NodeRacer, Shaker and FlakeShovel specifically detect concurrency-related flakiness. NodeRacer (Endo and Møller, 2020), analyses JavaScript programs and accelerates the manifestation of event races that can cause test flakiness. It uses instrumentation and builds a model consisting of a happens-after relation for callbacks. During the guided execution phase, this relation is used to explore postponing events such that callback interleaving is realistic with regard to actual executions. Shaker (Silva et al., 2020) suggests that the tool exposes flakiness faster than rerun by adding noise to the environment in the form of tasks that also stress the CPU and memory whilst the test suite is executed. FlakeShovel (Dong et al., 2021a) targets the same type of cause

Table 6
Flaky tests detection tools as reported in academic studies.

Detection type	Category	Language	Method type	Method	Tool name	Article
Outcomes	Order	Java	Dynamic	Rerun (Vary orders)	–	Wei et al. (2021)
Outcomes	Android	Java	Dynamic	Rerun (Vary event schedules)	FlakeScanner	Dong et al. (2021b)
Outcomes	General	Java	Dynamic	Rerun (twice)	–	Wei et al. (2022)
Cause	Web	Java	Dynamic	Rerun (Vary environment)	FlakyLoc	Morán Barbón et al. (2020)
Location	General	–	Dynamic	Log analysis	RootFinder	Lam et al. (2019a)
Outcomes	UI	Java	Dynamic	Rerun (Vary event schedules)	FlakeShovel	Dong et al. (2021a)
Outcomes	General	Java	Hybrid	Machine learning	FlakeFlagger	Alshammari et al. (2021)
Outcome	General	Mixed	Dynamic	Rerun (Environment)	–	Eloussi (2015)
Outcomes	General	Java	Static	Machine learning	–	Ahmad et al. (2020)
Outcomes	ML	Python	Dynamic	Rerun (Vary random number seeds)	FLASH	Dutta et al. (2020)
Outcomes	Concurrency	JavaScript	Dynamic	Rerun (Vary event order)	NodeRacer	Endo and Møller (2020)
Outcomes	General	Java	Static	Machine learning (test code similarity)	FLAST	Verdecchia et al. (2021)
Outcomes	General	Python	Dynamic	Rerun (Vary test code)	FITTER	Parry et al. (2020)
Outcomes	Concurrency	Java	Dynamic	Rerun (Add noise to environment)	Shaker	Silva et al. (2020)
Outcomes	General	Python	Dynamic	Test execution history	GreedyFlake	Vaidhyam Subramanian et al. (2020)
Outcomes	General	Java	Dynamic	Rerun	iDFlakies	Lam et al. (2019b)
Location	Assumptions	Java	Dynamic	Rerun (vary API implementation)	NonDex	Gyori et al. (2016)
Cause	Order	Java	Dynamic	Rerun and delta debugging	iFixFlakies	Shi (2020)
Outcomes	General	Java	Dynamic	Differential coverage and test execution history	DeFlaker	Bell et al. (2018)
Cause, location	General	C++/Java	Dynamic	Rerun	Flakiness Debugger	Ziftci and Cavalcanti (2020)
	UI	JavaScript	Dynamic	Machine learning (Bayesian network)	–	King et al. (2018)
Cause	Order	Java	Dynamic	–	PolDet	Gyori et al. (2015)
Outcome	General	–	Static	Machine learning	Flakify	Fatima et al. (2021)
Cause,Outcome	IO/Concurrency /Network	–	Dynamic	Rerun in varied containers	–	Terragni et al. (2020)
Cause,Outcome	–	–	Dynamic	Rerun in varied containers	TEDD	Biagiola et al. (2019)
Cause,Outcome	–	C	Static	Dependency analysis	–	Schwahn et al. (2019)
Outcomes	Order	Java	Dynamic	Rerun (Dynamic dataflow analysis)	PRADET	Gambi et al. (2018a)
Outcomes	Order	Java	Dynamic	Rerun (Vary order)	DTDetector	Zhang et al. (2014)
Outcomes	Order	Java	Dynamic	Rerun (Dynamic dataflow analysis)	ElectricTest	Bell et al. (2015)
Outcome	Order and Async/Wait	Java	Static	Pattern matching	–	Person and Elbaum (2015)
Outcome	Order	Python	Dynamic	Rerun (varying orders)	iPFlakies	Wang et al. (2022)
Outcome	–	Multilanguage	Dynamic	Machine learning	Fair	Haben et al. (2021b)
Outcome	Order	Java	Static	Model checking	PolDet (JPF)	Yi et al. (2021)
Outcome	Nondeterminism	Java	Static	Type checking	Determinism Checker	Mudduluru et al. (2021)

as NodeRacer by similarly exploring different yet feasible event execution orders, but only for GUI tests in Android applications.

Several tools are built to detect order-dependent tests. In the case of iDFlakies (Lam et al., 2019b), which uses rerun by randomizing the order of their execution, it classifies flaky tests into two types: order-dependent and non-order dependent. In this category there are four more studies: DTDetector (Zhang et al., 2014), ElectricTest (Bell et al., 2015), PolDet (Gyori et al., 2015), and PRADET (Gambi et al., 2018a). DTDetector presents four algorithms to check for dependent tests, which are manifested in test outcomes: reversal of test execution order, random test execution order, the exhaustive bounded algorithm (which executes bounded subsequences of the test suite instead of trying out all permutations), and the dependence-aware bounded algorithm that only tests subsequences that have data dependencies. ElectricTest checks for data dependencies between tests using a more sophisticated check for data dependencies. While DTDetector checks for writes/reads to/from static fields, ElectricTest checks for changes to any memory reachable from static fields. PRADET uses a similar technique to check for data dependencies, but it also refines the output by checking for manifest dependencies, i.e., data dependence that also influences flakiness in test outcomes. Wei et al. (2021) used a systematic and probabilistic approach to explore the most effective orders for manifesting order-dependent flaky tests. Whereas tools such as PRADET and DTDetector explore randomized test orders, Wei et al. analyse the probability of randomized orders detecting flaky tests, and they propose an algorithm that explores consecutive tests to find all order-dependent tests that depend on one test.

Wei et al. (2022) discussed a class of flakiness due to shared state, non-idempotent-outcome (NOP) tests, which are detected by executing the same test twice in the same VM.

NonDex (Gyori et al., 2016) is the only tool we found that detects flakiness caused by implementation dependency. The class of such dependencies it detects is limited to dependencies due to assumptions developers make about underdetermined APIs in the Java standard libraries, for instance, the iteration order of data structures using hashing in the internal representation, such as Java's `HashMap`.

Several studies discussed machine learning approaches for flakiness prediction. Pontillo et al. (2021) studied the use of test and production code factors that can be used to predict test flakiness using classifiers. Their evaluation uses a logistic regression model. Haben et al. (2021a) reproduced a Java study (Pinto et al., 2020) with a set of Python programs to confirm the effectiveness of code vocabularies for predicting test flakiness. Camara et al. (2021b) is another replication of the same study that extends it with additional classifiers and datasets. Parry et al. (2022) evaluated static and dynamic features that are more effective as flakiness predictors than previous feature sets. Camara et al. (2021a) evaluated using test smells to predict flakiness.

Table 7 summarizes the list of flaky test tools as found in grey literature. The table shows that most flaky test tools are either part of existing CI services or testing frameworks/tools. The goal of most of these tools is to identify flaky tests based on variations in test outcomes across multiple runs (i.e., observing changes from *FAIL* to *PASS*). Some CI-based tools also provide mechanisms to automatically quarantine tests that show variations of results across runs. Examples of CI services that provide flaky tests management functionality (e.g., as a dedicated dashboard) are CircleCI (Introducing test, 2021), Azure Pipelines (Manage flaky tests, 2020) and Datadog CI (Datadog, 0000). Some 'specialized' flaky test management tools provide tracking tools to manage flaky tests in production pipelines. These systems can help identify and track flaky tests found in test suites and warn developers of potential 'technical debt' related to their flaky tests. In our review sample, we identified two tools in this category: BuildPluse.[3] and Flalybot[4] Those tools can also be integrated into existing CI pipelines.

In terms of detection techniques, the most common approach used in these tools is rerun — in particular, retrying the failing tests multiple times to see if their outcomes will change (i.e., from FAIL to PASS). Several libraries are specifically designed to automatically rerun failed tests - i.e., to confirm the tests did not fail due to flakiness. Examples are pytest-flakefinder[5] and pytest-rerunfailures[6] (Flaky tests, 0000a) (both are pytest plugins) for Python, rspec-retry[7] (Mayer, 2019) for Ryby, protractor-flake[8] for JavaScript..

Another detection tool category targets specific flakiness causes. Such tools are designed to tackle specific causes of flaky tests in specific platforms, environments or programming languages. For example, `timecop`[9] (Tse, 2016; Shatrov, 2016) is a Ruby gem (library) that identifies time-dependent tests and provides a method to mock time calls to eliminate tests against time-related flakiness. Similarly, `pytest-random-order`[10] and `pytest-randomly`[11] are `pytest` extensions to detect order-dependent tests in Python by randomizing test orders. Capybara (Hoa, 2015) web testing framework provides an option to prevent test failures caused by race conditions (i.e., timeout wait time is customized by the user).

> **RQ2 summary.** Several methods have been proposed to detect flaky tests, which include static, dynamic and hybrid methods. Most static approaches use machine learning. Rerun (in different forms) is the most common dynamic approach for detecting flaky tests. Approaches that use rerun focus on making flaky test detection less expensive by accelerating ways to manifest flakiness or running fewer tests.

4.4. Flaky tests datasets (RQ1 and RQ2)

Datasets used in flakiness related studies can be divided into those used in empirical studies or for detection/causes analysis studies. Those are all obtained from academic literature studies. Table 8 lists these datasets with the type of flakiness in the programs, the programming language, the number of flaky tests identified and the total number of projects along with the names of the tools or if it is an empirical study.

As can be seen in Table 8, the dominant programming language is Java. There are a few studies in Python (Dutta et al., 2020). Some of these datasets are used in multiple studies, for instance (Lam et al., 2020d) obtains its subjects from Lam et al. (2019b), and Palomba and Zaidman (2017) from Bell et al. (2018).[12]

4.5. Impact of flaky tests (RQ3)

Next, we explore the wider view of the impact flaky tests have on different aspects of software engineering. In addressing this research question, we look at the impact of flaky tests, as discussed in the articles we reviewed, and then combine the evidence noted in academic and grey literature. We discuss this in detail in the following two subsections.

4.5.1. Impact noted in academic research

For each article we included in our review, we look at the context of flaky tests in the study. We classify the impact of flaky tests as reported in academic literature into the following three categories:

1. **Testing (including testing techniques):** the impact on software testing process in general (i.e., impact on test coverage).
2. **Product quality:** impact on the software product itself, and its quality.
3. **Debugging and maintenance:** the impact on other software development and program analysis techniques.

Fig. 7 illustrates these categories and provides a general taxonomy of impact points as noted in the reviewed studies. Table 9 shows a summary of the impact of flaky tests as noted in academic literature. We discuss some examples for each of the three categories below.

Impact on testing: Aspects of testing affected by test flakiness include the reliability and completeness of testing techniques, debugging and maintenance of tests, automated test generation and test optimization techniques. Software testing techniques rely on the assumption that tests have deterministic outcomes. They also rely on the test independence assumption, which states that tests, when executed independently (out of order or in isolation), should result in the same outcome as when they are executed as part of a test suite.

Automatic test generation tools (e.g. such as Randoop, EvoSuite) automatically generate entire test suites for detecting faults in the program and for regression testing. These tools may generate tests that are flaky (Fan, 2019) and hence break the assumptions for reliable and efficient testing. In addition, test optimization techniques such as test suite reduction, test prioritization, test selection, and test parallelization also rely on the same test reliability assumption. Test suite and test case reduction (MacIver and Donaldson, 2020; Shi et al., 2018) removes redundant test cases or minimizes tests to make regression testing more efficient. Removal of those tests can result in changes in test outcomes (e.g. due to test dependencies and order). Test selection, parallelization and prioritization (Lam et al., 2020b) techniques execute tests separately (as a subset of the test suite or in parallel), which can manifest flakiness. For instance, flakiness can manifest in order-dependent tests when test optimization is applied to test suites with such tests. Lam et al. (2020b) studied the necessity of dependent-test-aware techniques to reduce flaky test failures, where they first investigated the impact of flaky tests on three regression testing techniques: test prioritization, test selection and parallelization.

3 https://buildpulse.io/.
4 https://www.flakybot.com/.
5 https://pypi.org/project/pytest-flakefinder/.
6 https://pypi.org/project/pytest-rerunfailures/.
7 https://github.com/NoRedInk/rspec-retry.
8 https://www.npmjs.com/package/protractor-flake.
9 https://github.com/travisjeffery/timecop.
10 https://pypi.org/project/pytest-random-order/.
11 https://pypi.org/project/pytest-randomly/.

12 A dataset on the relationship between test smells and flaky tests was largely used in multiple studies but recently was retracted https://ieeexplore.ieee.org/document/8094404.

Table 7
Flaky tests tools as reported in grey literature.

Tool name	Type	Features	Availability	Method	Language	Reference
FlakyBot	CI	Determines test(s) are flaky before merging commits. The tools can be invoked on a pull request, and tests will be exercised quickly and results reported	Commercial	Rerun	Multi-language	Palmer (2019)
Azure DevOps Services	CI	Feature that enables the detection of flaky tests and keeping track of all flaky tests	Commercial	Rerun and changes diff	Multi-language	Manage flaky tests (2020)
Scope	Testing tool	Helps identify flaky tests, requiring a single execution based on the commit diff	Commercial	Track flaky commits	Multi-language	Lee (2020)
Cypress	Testing framework (web)	Automatically rerun (retries) a failed test prior to marking it as fail	Free	Rerun (fixed number of retries)	General	Rustamzadeh (2020)
Gradle Enterprise	Build tool	Considers a test flaky if it fails and then succeeds within the same Gradle task	Commercial	Track flaky commits	Multi-language	Wendelin (2019)
pytest-flakefinder & pytest-rerunfailures	Testing framework (library)	Rerun failing tests multiple times without having to restart pytest (in Python)	Free - Open source	Rerun (retry failed tests)	Python	Flaky tests (0000a)
pytest-random-order & pytest-randomly	Order-dependency detection	Randomize test order so that it can detect flakiness due to order dependency and expose tests with shared state problems	Free - Open source	Rerun (vary orders)	Python	Flaky tests (0000a)
BuildPluse	Testing tool	Detect and categorize flaky tests in the build by checking changes in test outcomes between builds	Commercial	Unknown	Multi-language	Detect (2020)
rspec-retry	Testing framework (library)	Ruby scripts that rerun flaky `RSpec` tests and obtain a success rate metric	Free - Open source	Rerun (vary order)	Ruby	Mayer (2019)
Quarantine	Testing framework (library)	A tool that provides a run-time solution to diagnosing and disabling flaky tests and automates the workflow around test suite maintenance	Free - Open source	Rerun (retry failed tests)	Ruby	Zhu (0000)
protractor-flake	Testing framework (library)	Rerun failed tests to detect changes in test outcomes	Free - Open source	Rerun (retry failed tests)	JavaScript TypeScript	Automated testing (2017)
Shield34	Testing tool (Web)	Designed to identify and quarantine Selenium flaky tests	Commercial	Unknown	Multi-language	Andrawis (2020)
Bazel	Build tool	An option to mark tests as flaky, which will skip those marked tests after 3 reruns	Free - Open source	Rerun (tests marked as flaky)	Multi-language	McCrary (2020) and Chodorow (2015)
Flaky	Testing framework (library)	Nose and pytest and nose to automatically rerunning failing tests	Free - Open source	Rerun (retry failed tests)	Python	Meadows (2014) and Strategies for handling (2019)
Capybara	Testing framework (web)	A test automation tool with an option to prevent against race conditions	Free - Open source	Rerun (Vary event schedules)	Multi-language	Hoa (2015)
Xunit.SkippableFact	Testing framework (library)	Tests can be marked as SkippableFact allowing control over test execution	Free - Open source	Sanitization (avoid running a test under certain conditions)	C#	Richter (2020)
timecop	Testing framework (library)	Ruby framework to test time-dependent tests	Free - Open source	Sanitization (controlling time-dependent test)	Ruby	Tse (2016) and Shatrov (2016)
Athena	Testing tool	Identifies commits that make a test nondeterministically fail, and notifying the author. It will also automatically quarantines flaky tests	Internal (Dropbox)	Unknown	Multi-language	Shah (2019)

(continued on next page)

Table 7 (*continued*).

Tool name	Type	Features	Availability	Method	Language	Reference
Datadog	CI	Flaky test management through a visualization of test outcomes.	Commercial	Unknown	Multi-language	Datadog (0000)
CircleCI dashboard	CI	The "Test Insights" dashboard provides information about all flaky tests, with an option to automate reruns of failed tests	Commercial	Track flaky commits		Introducing test (2021)
Flaky-test-extractor-maven-plugin	Build tool	Maven plugin that filters out flaky tests from existing surefire reports. It generates additional reports just for the flaky tests	Free - Open source	Sanitization (avoid running a test under certain conditions)	Java	flaky-test-extractor-maven-plugin (0000)
TargetedAutoRetry	Xcode IDE package	A tool to retry just the steps which are most likely to cause issues with flakiness (such as Apps launch, race conditions candidates etc..)	Free - Open source	rerun (vary events schedule)	Swift	eBay (2021)
JUnit surefire plugin	Testing framework	An option to rerun failing tests in Junit surefire plugin (rerunFailingTestsCount)	Free - Open source	Sanitization (avoid running a test under certain conditions)	Java	Maven (0000)
Gradle's Test Failure Analytics	Build tool	Gradle plugin that helps to identify flaky tests between different builds	Commercial	Track flaky commits	Multi-language	Gradle (0000)
Test Analyzer Service	Testing tool	A tool to manage the state of unit tests and to disable flaky tests	Internal (Uber)	Track flaky commits Sanitization	Multi-language	Agarwal (2021)
TestRecall	Testing tool	Test analysis tool that provides insights about test suites, including tracking flaky tests	Commercial	Track flaky commits	Multi-language	Klotz (0000)
Katalon Studio	Testing framework	An option to retry all tests (or only failed tests) when the Test Suite finishes	Commercial	Rerun (retry failed tests)	Multi-language	Katalon (0000)

Table 8
Datasets to study test flakiness.

Study	Flakiness type	Language	# flaky tests	# projects	Article
iDFlakies	Order dep/Other	Java	422	694	Lam et al. (2019b)
DeFlaker	General	Java	87	96	Bell et al. (2018)
NonDex	Wrong assumptions	Java	21	8	Gyori et al. (2016)
iFixFlakies	Order dependent	Java	184	10	Shi (2020)
FLASH	Machine learning	Python	11	20	Dutta et al. (2020)
Shaker	Concurrency	Java/Kotlin (Android)	75	11	Silva et al. (2020)
FlakeShovel	Concurrency	Java (Android)	19	28	Dong et al. (2021a)
NodeRacer	Concurrency	JavaScript	2	8	Endo and Møller (2020)
GreedyFlake	Flaky coverage	Python	–	3	Vaidhyam Subramanian et al. (2020)
Travis-Listener	Flaky builds	Mixed	–	22,345	Durieux et al. (2020)
RootFinder	General	.Net	44	22	Lam et al. (2019a)

Other testing techniques impacted by test flakiness are test amplification (Abdi et al., 2021), simulation (Ahlgren et al., 2021) and manual testing (Haas et al., 2021).

Impact on product quality: Several articles cite how test flakiness breaks builds (Vehabovic, 2020; Widder et al., 2019). Testing drives automated builds, which flakiness can break, resulting in delaying CI workflows. Zdun et al. (2019) highlighted how flaky tests can introduce noise into CI builds that can affect service deployment and operation (microservices and APIs in particular). Böhme (2019) discussed flakiness as one of the challenges for test assurance, i.e., executing tests as a means to increase confidence in the software. Product quality can be affected due to lack of test stability, cited as an issue by Hirsch et al. (2019), in the context of a single Android application with many fragile UI tests. Several articles mention the issue of cost in detecting flaky tests; Pinto et al. (2020) pointed out that it can be costly to run detectors after each change and hence organizations run them only on new or changed tests, which might not be the best approach as this would affect the recall. Vassallo et al. (2020) identified retrying failure to deal with flakiness as a CI smell, as it has a negative impact on the development experience by slowing down progress and hiding bugs. Mascheroni et al. (2021) proposed a model to improve continuous testing by presenting test reliability as a level in the improvement model, and flaky tests as a main cause for reliability issues with tests. They suggest good practices to achieve this.

Multiple articles also discuss how test flakiness can affect developers, negatively impacting product quality. This includes the developer's perception of tests, and the effort required to respond to events arising from test flakiness (build failures in CI, localizing causes, fixing faulty tests). Koivuniemi (2017) mentioned uncertainty and frustration due to developers attributing flaky failures to errors in the code where there are none. Eck et al. (2019a) survey on developer's perception of flaky tests noted that flaky tests could have an impact on software projects, in particular on resource allocation and scheduling.

Impact on debugging and maintenance: Several techniques used in maintenance and debugging are known to be impacted by the presence of flaky tests. This includes all techniques that rely on tests, such as test-based program repair, crash reproduction, test amplification and fault localization, which can all be negatively impacted by flakiness. Martinez et al. (2017) reported

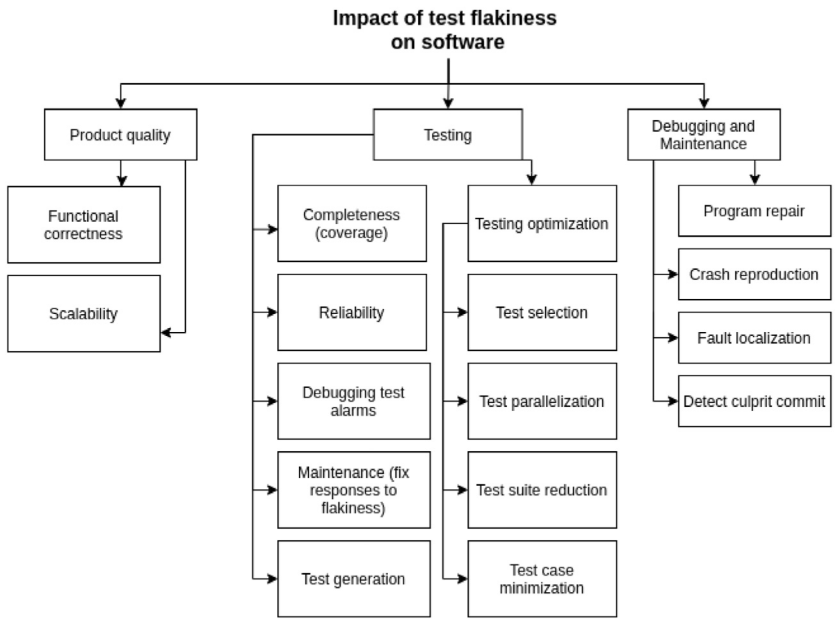

Fig. 7. Taxonomy for the impact of test flakiness.

a flaky test in a commonly used bugs' dataset, Defects4J, and how the repair system's effectiveness can be affected (if the flaky test fails after a repair, the system would conclude that the repair introduced a regression). Chen et al. (2020) explained that subpar quality tests can affect their use for detecting performance regressions, and in the case of flaky tests, they may introduce noise and require multiple executions. Dorward et al. (2021) proposed a more efficient approach to find culprit commits using flaky tests, as bisect fails in this situation.

4.5.2. Impact noted in grey literature

We also analysed the impact of flaky tests as found in grey literature articles. We checked if there was any discussion of the impact of flaky tests on certain techniques, tools, products or processes. We classify the noted impact of flaky tests into the following three categories:

1. **Code-base and product:** the impact of flaky tests on the quality or the performance of the production code and the CUT.
2. **Process:** the impact on the development pipeline and the delivery of the final product.
3. **Developers:** the 'social' impact of flaky test on the developers/testers.

A summary of the impact noted in the grey literature is shown in Table 10. We discussed each of those three categories below.

Impact on the code-base and product: Several grey literature articles have discussed the wider impact of flaky tests on the production code and on the final software product. Among several issues reported by different developers, testers and managers, it was noted that the presence of flaky tests can significantly increase the cost of testing (Lee, 2020; Zhu, 0000), and makes it hard to debug and reproduce the CUT (Eloussi, 2016; Otonelli, 2017; Saffron, 2019). In general, flaky tests can be very expensive to repair and often require time and resources to debug (Saffron, 2019; Flaky tests, 2021). They can also make end-of-end testing useless (Grant, 2016), which can reduce test reliability (Managing test flakiness, 0000; Listfield, 2017). One notable area that flaky tests compromise is coverage — if the test is flaky enough that it can fail even when retried, then coverage is already considered lost (A pragmatist, 2021). Flaky tests can also spread and accumulate, with some unfixed flaky tests can lead to more flaky tests in the test suite (Strĭĉeviĉ, 2015; Lee, 2020). Fowler described them as *"virulent infection that can completely ruin your entire test suite"* (Fowler, 2011a).

Flaky tests can have serious implications in terms of time and resources required to identify and fix potential bugs in the CUT, and can directly impact production reliability (Zhu, 0000). However, detecting and fixing flaky tests can help in finding underlying flaws and issues in the tested application and CUT that is otherwise much harder to detect (Saffron, 2019).

Impact on developers: We observed that several blog posts we analysed here are written by developers and discuss the impact of flaky tests on their productivity and confidence. Developers noted that flaky tests can cause them to lose confidence in the 'usefulness' of the test suite in general (Palmer, 2019), and to lose trust in their builds (Grant, 2016). Flaky tests may also lead to a "collateral damage" for developers: if left uncontrolled or unresolved, they can have a bigger impact and may ruin the value of an entire test suite (Lee, 2020). They are also reported to be disruptive and counter-productive, as they can waste developers' time as they try to debug and fix those flaky tests (Saffron, 2019; Micco, 2017; Sudarshan, 2012; Yarn, 2016).

> *"The real cost of test flakiness is a lack of confidence in your tests..... If you do not have confidence in your tests, then you are in no better position than a team that has zero tests. Flaky tests will significantly impact your ability to confidently [and] continuously deliver."* (*Spotify Engineering*, Palmer (2019)).

Another experience report from Microsoft explained the practices followed, and tools used to manage flaky tests at Microsoft in order to boost developers' productivity:

> "*Flaky tests.... negatively impact developers' productivity by providing misleading signals about their recent changes ... developers may end up spending time investigating those failures, only to discover that the failures have nothing to do with their changes and may simply go away by rerunning the tests*". (*Engineering@Microsoft*, Improving developer (2022))

Table 9
Summary of the impact of flaky tests noted in academic literature.

Impact type	Impact	Reference
Product quality	Breaking builds	Vehabovic (2020) and Widder et al. (2019)
	Service deployment and operation	Zdun et al. (2019)
	Test reliability	Mascheroni et al. (2021)
	Test assurance	Böhme (2019)
	Product quality	Hirsch et al. (2019)
	Costly to detect	Lam et al. (2020d), Pinto et al. (2020) and Chan et al. (2007)
	Delays CI workflow	Vassallo et al. (2020) and Durieux et al. (2020)
	Maintenance effort	Lassila et al. (2019)
	Uncertainty and frustration	Koivuniemi (2017)
	Trust in tools and perception	Cox and Chen (2019) and Candido et al. (2017)
Testing	Regression testing techniques	Lam et al. (2020b)
	Simulation testing	Ahlgren et al. (2021)
	Test amplification	Abdi et al. (2021)
	Test suite/case reduction	Shi et al. (2018) and MacIver and Donaldson (2020)
	Mutation testing	Demeyer et al. (0000) and Laurent et al. (2020)
	Manual testing	Haas et al. (2021)
	Test minimization	Vahabzadeh et al. (2018)
	Test coverage (ignored tests)	Gabrielova (2017)
	Test selection	Elbaum et al. (2014) and Mansky and Gunter (2020)
	Patch quality	Ginelli et al. (2020)
	Test performance	Ramler et al. (2019)
	Test suite efficiency	Alvaro et al. (2012)
	Test prioritization	de Oliveira Neto et al. (2020) and Elbaum et al. (2014)
	Regressions	Paydar and Azamnouri (2019)
	Test suite diversity	de Oliveira Neto et al. (2020)
	Test generation	Linares-Vásquez et al. (2017)
	Differential testing	Malm et al. (2020)
	Test assurance	Böhme (2019)
Debugging and maintenance	Program repair	White et al. (2019), Martinez et al. (2017) and Ye et al. (2021)
	Determining culprit commits	Dorward et al. (2021)
	Performance analysis	Chen et al. (2020)
	Bug reproduction	Urli et al. (2018)
	Crash reproduction	Soltani et al. (2020)
	Fault localization	Vancsics et al. (2020) and Jeon and Hong (2020)

Table 10
Summary of the impact of flaky tests as noted in grey literature.

Impact type	Impact	Reference
Product	Hard to debug	Eloussi (2016), Otonelli (2017) and Saffron (2019)
	Hard to reproduce	Eloussi (2016)
	Reduces test reliability	Managing test flakiness (0000) and Listfield (2017)
	Expensive to repair	Flaky tests (2021)
	Increase cost of testing as flaky behaviour can spread to other tests	Lee (2020) and Tips on treating (2017)
Developers	Losing trust in builds	Grant (2016), Tse (2016), Flaky tests (2021), How (2020) and Katalon (0000)
	Loss of productivity	Lee (2020), 10 reasons (2015), Fixing a flaky test (0000) and Tips on treating (2017)
	Time-consuming/wastes time	Wendelin (2019), Saffron (2019) , Micco (2017), Improving developer (2022), Gradle (0000), TestProject (2021), Reducing flaky (2020) and Agarwal (2021)
	Resource consuming	Sudarshan (2012), Yarn (2016) and How (2020)
	Demotivate/mislead developers	Wendelin (2019) and Improving developer (2022)
Delivery	Affects the quality of shipped code	Manage flaky tests (2020), A pragmatist (2021) and Klotz (0000)
	Slows down deployment pipeline	Wendelin (2019), Saffron (2019), Flaky tests (2021), Gradle (0000) and How to deal (2021)
	Slows down the development	Peterson (2019), Saffron (2019), Testinium (2018), Wendelin (2019) and Test flakiness (2020)
	Loses faith in tests catching bugs	Yarn (2016) and Sandhu (2015)
	Causes unstable deployment pipelines	Mayer (2019)
	Slows down development and testing processes	Peterson (2019) and Test flakiness (2020)
	Delays project release	Micco (2017), Shah (2019) and eBay (2021)

Impact on delivery: Developers and managers also presented evidence of how flaky tests can delay developments and have a wider impact on the delivery (e.g., Zhu (0000)) - mostly by slowing down the development (Peterson, 2019; Saffron, 2019; Testinium, 2018) and delaying products' release (Micco, 2017; Shah, 2019). They can also reduce the value of an automated regression suite (Fowler, 2011a) and lead organization and testing teams to lose faith that their tests will actually find bugs (Yarn, 2016; Sandhu, 2015). Some developers also noted that if flaky tests are left unchecked or untreated, they can lead to completely useless test suites, as this is the case with some organizations:

"*We've talked to some organizations that reached 50%+ flaky tests in their codebase, and now developers hardly ever write any tests and do not bother looking at the results. Testing is no longer a useful tool to improve code quality within that organization*". (*Product Manager at Datadog*, Lee (2020))

Another impact of flaky tests is that they could slow down the deployment pipeline which can decrease confidence in the correctness of changes in the software (Wendelin, 2019; Flaky tests, 2021). They could even block deployment until spotted and resolved (Flaky tests, 2017).

RQ3 summary. The impact of flaky tests has been the subject of discussion in both academic and grey literature. Flaky tests are reported to have an impact on the products under development, the quality of CUT and the tests themselves and the delivery pipelines. Techniques that rely on tests such as test-based program repair, crash reproduction and fault detection and localization can be negatively impacted by the presence of flaky tests.

4.6. Responses to flaky tests (RQ4)

The way that developers and teams respond to flaky tests has been discussed in detail in both academic and grey literature. However, the type of applied/recommended response is slightly different from one study to another as this also depends on the context of the causes of flaky tests, and also the methods used to detect them. Below we discuss the responses as noted in academic and grey literature, separately:

4.6.1. Response noted in academic literature

We classify responses to flaky tests as follows:

- Modifying the test.
- Modifying the program/code under test.
- Process response.

We provide a general taxonomy of the responses to flaky tests as noted in the reviewed studies in Fig. 8.

A summary of the responses found in academic articles is presented in Table 11. The three major strategies are to fix tests, modify the CUT or put in a mechanism to deal with flaky tests (e.g., retry or quarantine tests). Berglund and Vateman (2020) listed some strategies for avoiding non-deterministic behaviour in tests: minimizing variations in the testing environment, avoiding asynchronous implementations, testing in isolation, aiming for deterministic assertions and limiting the use of third-party dependencies. Other measures include mocking to reduce flakiness, for instance, EvoSuite (Fraser and Arcuri, 2011) uses mocking for this. Zhu et al. (2020) proposed a tool for identifying and proposing mocks for unit tests. A wider list of specific fixes to the different types of flaky tests is provided in Luo et al. (2014). Shi et al. (2019) presented a tool, iFixFlakies, to fix order-dependent tests.

Fixes in the CUT are not discussed as much in academic articles. The closest mention in relation to this is in Thorve et al. (2018), which finds instances in flaky test fix commits where the CUT is improved and dependencies are changed to fix flakiness.

Another strategy, removing flaky tests, was also identified in Thorve et al. (2018). The study found that developers commented out flaky tests in 10/77 of examined commits. Removing flaky tests is also a strategy cited in papers that discuss testing-related techniques (Gay, 2017; Danglot et al., 2019). Quarantining, ignoring or disabling flaky tests are also discussed as responses.

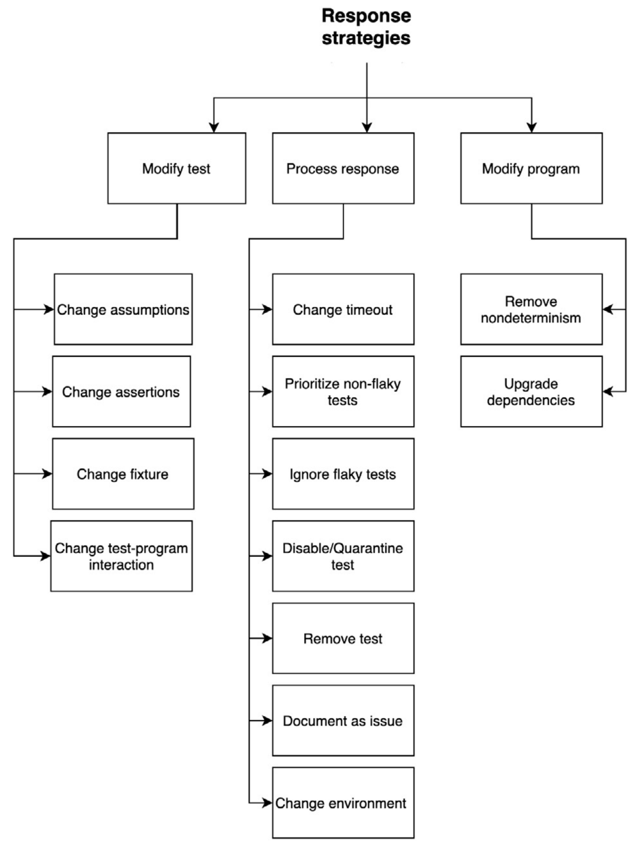

Fig. 8. Taxonomy of response strategies.

Memon et al. (2017) detailed the approach at Google for dealing with flaky tests. They use multiple factors (e.g., more frequently modified files are more likely to cause faults) to prioritize the tests to rerun rather than a simple test selection heuristic such as rerun tests that failed recently, which is sensitive to flakiness.

A number of tools have been proposed recently for automatically repairing flaky tests. They can fix flakiness due to these causes: randomness in ML projects, order dependence and implementation dependence. Haas et al. (2021) and Dutta et al. (2022) conducted an empirical analysis of seeds in machine learning projects and propose approaches to repair flaky tests due to randomness by tuning hyperparameters, fixing seeds and modifying assertions bounds. Zhang et al. (2021) proposed a tool for fixing flaky tests that are caused by implementation dependencies of the type explored by NonDex (Gyori et al., 2016). Wang et al. (2022) proposed iPFlakies for Python that fixes order-dependent tests that fail due to state that is polluted by other tests. This is related to their work, iFixFlakies for repairing order-dependent tests in Java programs. The Python tool discovers existing tests or helper methods that clean the state before successfully rerunning the order-dependent test. ODRepair from Li et al. (2022) is an approach that uses automatic test generation to clean the state rather than existing code. Mondal et al. (2021) proposed an approach to fixing flakiness due to parallelizing dependent tests by adding a test from the same class to correct dependency failure.

4.6.2. Response noted in grey literature

Here we look at the methods and strategies followed to deal with flaky tests as noted in the grey literature. We classified those strategies into the following categories:

Table 11
Summary of the response strategies to flaky tests in academic literature.

Strategy	Description	Articles
Modify test	**Change assumptions**	
	Fix assumptions about library API's	Gyori et al. (2016)
	Automatically repair implementation-dependent tests	Zhang et al. (2021)
	Replace test	Eck et al. (2019a)
	Merge dependent tests	Luo et al. (2014)
	Change assertions	
	Modify assertion bounds (e.g., to accommodate wider ranges of outputs)	Luo et al. (2014), Thorve et al. (2018), Eck et al. (2019a), Dutta et al. (2020), Haas et al. (2021) and Dutta et al. (2022)
	Change fixture	
	Removing shared dependency between tests	Luo et al. (2014)
	Global time as system variable	Silva et al. (2019)
	Setup/clean shared state between tests	Luo et al. (2014)
	Modify test parameters	Dutta et al. (2020)
	Modify test fixture	Shi et al. (2019)
	Fix defective tests	Wild et al. (2020), Grano et al. (2019) and Luo et al. (2014)
	Make behaviour deterministic	Berglund and Vateman (2020)
	Change delay for async-wait	Malm et al. (2020) and Olianas et al. (2021)
	Concurrency-related fixes	Eck et al. (2019a)
	Change test-program interaction	
	Mock use of environment/concurrency	Zhu et al. (2020)
Modify program	Concurrency-related fixes	Luo et al. (2014)
	Replace dependencies	Thorve et al. (2018) and Dutta et al. (2020)
	Remove nondeterminism	Eck et al. (2019a)
Process response	Rerun tests	Thorve et al. (2018), Ginelli et al. (2020) and Laurent et al. (2020)
	Ignore/Disable	Eck et al. (2019a), Beller et al. (2021) and Chan et al. (2007)
	Quarantine	Storey and Zagalsky (2016)
	Add annotation to mark test as flaky	Dutta et al. (2020)
	Increase test time-outs	Eck et al. (2019a)
	Reconfigure test environment (e.g., containerize or virtualize unit tests)	Bell and Kaiser (2014)
	Remove	Gay (2017), Tomassi and Rubio-González (2019), Berglund and Vateman (2020), Danglot et al. (2019) and Mascheroni et al. (2019)
	Improve bots to detect flakiness	Erlenhov et al. (2020)
	Responsibility of CI to deal with it	Machalica et al. (2019)
	Prioritize tests	Memon et al. (2017) and Rehman and Rigby (2021)

1. **Quarantine:** keep flaky tests in a different test suite to other 'healthy' tests in a quarantined area in order to diagnose and then fix those tests.
2. **Fix immediately:** fix any flaky test that has been found immediately, but first developers will need to reproduce the flaky behaviour.
3. **Skip and ignore:** provide an option to developers to ignore flaky tests from the build and suppress the test failures. This is usually in the form of annotation. In some cases, especially when developers are fully aware of the flaky behaviour of the tests and the implications of those tests have been considered, they may decide to ignore those flaky tests and continue with the test run as planned.
4. **Remove:** remove any test that is flaky from the test suite once detected.

A summary of the response found in grey literature is shown in Table 12. The most common strategy that has been discussed is to quarantine and then fix flaky tests. As explained by Fowler (2011a), this strategy indicates that developers should follow a number of steps once a flaky test has been identified: *Quarantine → Determine the cause → Report/Document → Isolate and run locally → Reproduce → Decide (fix/ ignore)*. This is the same strategy that Google (and many other organizations) has been employing to deal with any flaky tests detected in the pipelines (Micco, 2016). A report from Google reported that they use a tool that monitors all potential flaky tests, and then automatically quarantines the test in case flakiness is found to be high. The quarantining works by removing "*the test from the critical path and files a bug for developers to reduce the flakiness. This prevents it from becoming a problem for developers, but could easily mask a real race condition or some other bug in the code being tested*" (Micco, 2016). Other organizations also follow the same strategy e.g., Flexport (Zhu, 0000) and Dropbox (Shah, 2019). Flexport (Zhu, 0000) have even included a mechanism to automate the process of quarantining and skipping flaky tests. The Ruby gem, Quarantine,[13] is used to maintain a list of flaky tests by automatically "detects flaky tests and disables them until they are proven reliable".

It has been suggested by some developers and managers that all identified flaky tests should be labelled by their severity. This can be determined by which specific component they impact, the frequency of a flaky test, or the flakiness rate of a given test. One approach that has been suggested is not only to quarantine and treat all flaky tests equally but to quantify the level of flakiness of each flaky test so that those tests can be priorities for fixing. A report from Facebook engineers proposed a statistical metric called the Probabilistic Flakiness Score (PFS), with the aim to quantify flakiness by measuring test reliability based on how flaky they are (How, 2020). Using this metric, developers can "*test the tests to measure and monitor their reliability, and thus be able to react quickly to any regressions in the quality of our test suite. PFS ... quantify the degree of flakiness for each individual test at Facebook and to monitor changes in its reliability over time. If we detect specific tests that became unreliable soon after they were created,*

[13] https://github.com/flexport/quarantine.

Table 12
Summary of the response strategies followed by some organizations to deal with flaky tests, as discussed in grey literature.

Strategy	Description	Example
Quarantine	Keep flaky tests in a different test suite to other healthy tests in a quarantined area.	Micco (2016), Fowler (2011a), Flaky tests (2017), Zhu (0000), SamGu (2020), Lapierre (2017) and Shah (2019) Mayer (2019), Hallam (2019), Strategies for handling (2019), Richter (2020), McPeak (2018), Sandhu (2015), Flaky tests (0000b) and Flaky tests (2021) Agarwal (2021), How to deal (2021), Testing in Chromium (0000), eBay (2021), Improving developer (2022) and Klotz (0000)
Fix and replace immediately, or remove if not fixed	Test with flaky behaviour are given priority and fixed/removed once detected.	Rushakoff (2019), Stosik (2020), Rakiĉ (2017), Raposa (2020), Reducing flaky (2020), How to debug (2019) and Flaky tests (0000b)
Label flaky tests	Leave it to developers to decide	Wendelin (2019), Strategies for handling (2019), A pragmatist (2021), Improving developer (2022) and Klotz (0000)
Ignore/Skip	Provide an option to developers to ignore flaky tests from the build (e.g., though the use of annotations) and suppress the test failures.	Manage flaky tests (2020), Lee (2020), Richter (2020) and A pragmatist (2021)

we can direct engineers' attention to repairing them." How (2020). GitHub reported a similar metrics-based approach to determine the level of flakiness for each flaky test. An impact score is given to each flaky test based on how many times it changed its outcomes, as well as how many branches, developers, and deployments were affected by it. The higher the impact score, the more important the flaky test and thus the highest priority for fixing is given to this test (Reducing flaky, 2020).

At Spotify (Palmer, 2019), engineers use Odeneye, a system that visualizes an entire test suite running in the CI, and can point out developers to tests with flaky outcomes as the results of different runs. Another tool used at Spotify is Flakybot,[14] which is designed to help developers determine if their tests are flaky before merging their code to the master/main branch. The tool can be self-invoked by a developer in a pull request, which will exercise all tests and provide a report of their success/failure and possible flakiness.

There are a number of issues to consider when quarantining flaky tests though, such as how many tests should be quarantined (having too many tests in the quarantine can be considered as counterproductive) and how long a test should stay in quarantine. Fowler (2011a) suggested that not more than 8 tests be in the quarantine at one time, and not to keep those tests for a long period of time. It was suggested to have a dashboard to track the progress of all flaky tests so that they are not forgotten (Lee, 2020), and have an automated approach, not to only quarantine flaky tests, but also to de-quarantine them once fixed or decided to be ignored (Klotz, 0000).

Regarding the different causes of flaky tests, there are different strategies that are recommended to deal with the specific sources of test flakiness. For example, to deal with flakiness due to state-dependent scenarios such as if there is an "Inconsistent assertion timing" (i.e., state is not consistent between test runs that can cause tests to fail randomly), one solution is to "construct tests so that you wait for the application to be in a consistent state before asserting" (Zhu, 0000). If the test depends on specific test order (i.e., global state shared between tests as one test may depend on the compilation of another one), an obvious solution is to "reset the state between each test run and reduce the need for global state" (Zhu, 0000). Table 13 provides a brief summary of flaky tests' fixing strategies due to the most common causes as noted in grey literature articles.

[14] https://www.flakybot.com.

RQ4 summary. Quarantining flaky tests (for a later investigation and fix) is a common strategy that is widely used in practice. This is now supported by many tools that can integrate with modern CI tooling (able to automatically detect changes in test outcomes to identify flaky tests). Understanding the main cause of the flaky behaviour is key to reproducing flakiness and identifying an appropriate fix, which remains a challenge.

5. Discussion

In this section, we discuss the results of the review and present possible challenges in detecting and managing flaky tests. We also provide our own perspective on existing approaches' limitations and discuss potential future research directions.

5.1. Flaky tests in research and practice

The problem with flaky tests has been widely discussed among researchers and practitioners. Dealing with flaky tests is a real issue that is impacting developers and test engineers on a daily basis. It can undermine the validity of test suites and make them almost useless (Lee, 2020; Grant, 2016). The topic of flaky tests has been a research focus, with a noticeable increase in the number of publications over the last four years (between 2017 and 2021). We observed that the way the issue of test flakiness is being discussed is slightly different in academic and grey literature. Most research articles discuss the impact of flaky tests on different software engineering techniques and applications. Research that focuses solely on flaky tests mainly tackles new methods that are employed to detect flaky tests, aiming to increase speed (i.e., how fast flakiness can be manifested) and the accuracy of flakiness detection. Many of those studies have focused on specific causes of flakiness (either in terms of detection or fixes) - namely those related to order-dependency in test execution or to concurrency. There is generally a lack of studies that investigate the impact of other causes of test flakiness, such as those related to variation in the environment or in the network. This is an area that can be addressed by future tools designed specifically to detect test flakiness due to those factors. Our recent work targets this by designing a tool, *saflate*, that is aimed at reducing test flakiness by sanitizing failures induced by network connectivity problems (Dietrich et al., 2022).

On the other hand, the discussion in grey literature focused more on the general strategies that are being followed in practice to deal with any flaky tests once detected in the CI pipeline.

Table 13
Some fixing strategies for some common flaky tests noted in grey literature.

Cause of flakiness	Suggested fix	Example
Asynchronous wait	Wait for a specified period of time before it checks if the action has been successful (with callbacks and polling).	Strîĉeviĉ (2015) and Grant (2016)
Inconsistent assertion timing	Construct tests so that you wait for the application to be in a consistent state before asserting.	Strîĉeviĉ (2015)
Concurrency	Make tests more robust so that it accepts all valid results. Avoid running tests in parallel.	Strîĉeviĉ (2015) and Grant (2016)
Order dependency	Run a test in a database transaction that is rolled back once the test has finished executing. Clean up the environment (i.e., reset state) and prepare it before every test (and reduce the need for global state in general). Run test in isolation. Run test in random order to find out if they are still flaky.	Strîĉeviĉ (2015), Strîĉeviĉ (2015), Palmer (2019), Berczuk (2020), Flaky tests (2017), Stosik (2020) and Shatrov (2016)
Time-dependent tests	Wrapping the system clock with routines that can be replaced with a seeded value for testing. Use a tool to control for time variables such as freeze time helper in Ruby and Sinon.JS in JavaScript.	Flaky tests (2017) and Saffron (2019)
Randomization	Avoid the use of random seeds.	Hallam (2019)
Environmental	Limit dependency on environments in the test. limit calls to external resources and build a mocking server for tests.	Managing test flakiness (0000), Tse (2016) and Testinium (2018)
Leak global state	Run test in random order.	Saffron (2019)

Those are usually detected by checking if the test outcomes have changed between different runs (e.g., between PASS to FAIL). Several strategies that have been followed by software development teams are discussed in grey literature, especially around what to do with flaky tests once they have been identified. A notable approach is quarantining flaky tests in an isolated 'staging' area before they are fixed (Fowler, 2011a; Storey and Zagalsky, 2016).

The gap between academic research and practice when it comes to the way flaky tests are viewed has also been discussed in some of the most recent articles. An experience report published by managers and engineers at Facebook (How, 2020) explained how real-world applications can *always* be flaky (e.g., due to the non-determinism of algorithms), and what we should be focusing on is not when or if tests are flaky, but rather how flaky those tests can be. This supports Harman and O'Hearn's view (Harman and O'Hearn, 2018) that all tests should, by default, be considered flaky, providing a defensive mechanism that can help manage flaky tests in general.

5.2. Identifying and detecting flaky tests

In RQ1, we surveyed the common causes of flaky tests, whether in the CUT or in the tests themselves. We observe that there are a variety of causes for flakiness, from the use of specific programming language features to the reliance on external resources. It is clear that there are common factors that are responsible for flaky test behaviours, regardless of the programming language used or the application domains. Factors like test order dependency, concurrency, randomness, network and reliance on external resources are common across almost all domains and are responsible for a high proportion of flaky tests.

Beyond the list of causes noted in the first empirical study on this topic (Luo et al. (2022)), we found evidence of a number of additional causes, namely flakiness due to algorithmic nondeterminism (related to randomness), variations of hardware, environment and those related to the use of ML applications (which are nondeterministic in nature).

Some causes identified in academic literature overlap and causes can also be interconnected. For example, UI flakiness can, in turn, be due to a platform dependency (e.g. dependency on a specific browser) or because of event races.

With this large number of causes of flaky tests, further in-depth investigation into the different causes is needed to understand how flaky tests are introduced in the code base and better understand the root causes of flaky tests in general. This also includes studies of test flakiness in the context of a variety of programming languages (as opposed to Java or Python, which most flakiness studies have covered).

5.3. The impact of and response to flaky tests

It is clear that flaky tests are known to have a negative impact on the validity of the tests or the quality of the software as a whole. A few impact points have been discussed in both academic and grey literature. Notable areas that are impacted by flaky tests are test-dependent techniques, such as fault localization, program repair and test selection. An important impact area that has not been widely acknowledged is how flaky tests affect developers. Although the impact on developers was mentioned in developers' surveys (Eck et al., 2019a; Gruber and Fraser, 2022), and in many grey literature articles (e.g., Palmer (2019), Lee (2020) and Improving developer (2022)), such an impact has not been explicitly studied in more detail — an area that should be explored further in the future.

In terms of responses to flaky tests, it seems that the most common approach is to quarantine flaky tests once they are detected. The recommendation is to keep tests with flaky outcomes in a separate quarantine area from other "healthy" tests. This way, those flaky tests are not forgotten, and the cause of flakiness can be investigated later to apply a suitable fix. On the other hand, other non-flaky tests can still run so that it does not cause any delay in development pipelines. However, some open questions remain about how to deal with quarantined tests, how long those tests should stay in the designated quarantine area, and how many tests can be quarantined at once. A strategy (that can be implemented into tools) for processing quarantined flaky tests and removing them from the designated quarantine area (i.e., de-quarantining) also needs further investigation.

One interesting area for future research is to study the long-term impact of flaky tests. For example, what is the impact of flaky tests on the validity of test suites if left unfixed or unchanged? Do a few flaky tests that are left in the test suite untreated have a wider impact on the presence of bugs as the development progresses? It is also interesting to see, when flaky tests are flagged and quarantined, how long it will take developers to fix those tests. This can be viewed as a technical debt that will need to be paid back. Therefore, a study on whether this is actually being paid back, and how long it takes, will be valuable.

5.4. Comparing test flakiness in academic and grey literature context

With reviewing both academic and grey literature material, we noted a different focus and perspective on what flakiness means in the context of academic studies compared to what practitioners discuss in the grey literature posts. We discuss those differences below.

The first notable difference is the overall focus in academic and grey literature. Academic literature focuses mainly on detecting and locating flaky tests. Flaky test detection strategies rely on the underlying cause of flakiness. The surveyed academic studies provide an in-depth investigation of the common causes of flaky tests in different contexts (such as the programming language used, test environment and application domain). Many academic studies primarily focus on ways to manifest flaky tests, either by manipulating the tests, their interactions or their running environment, to reveal any potential changes in test behaviour and outcomes (such as studies on flakiness due test order-dependency Dong et al., 2021b; Wei et al., 2021; Parry et al., 2022).

In the surveyed grey literature, on the other hand, the focus is mostly on documenting and reporting flaky tests. The general point of discussion is how to track flaky tests, especially within the regression testing process, and how CI tools manage flaky tests regardless of the underlying cause. Identifying flakiness causes is a critical step towards dealing with flaky tests (e.g., if the tests can be fixed).

The second difference we observed is regarding the detection tool. Academic literature has presented tools that aim to detect flaky tests either statically or dynamically. The goal is to manifest flakiness in test suites. Thus, most of the presented tools are actually detection tools, with many focusing on the specific causes of flakiness (for example, flakiness due to test order Lam et al., 2019b or concurrency Silva et al., 2020). In grey literature, the tools we found focused mostly on tracking and reporting flaky tests, and the detection is done based on changes in test outcomes (especially for tests that suddenly fail). Those tools come as part of CI services (such as in CircleCI How to reduce, 2022) or existing testing tools (such as in Capybara Hoa, 2015).

The third difference is regarding response strategies. In academic literature, there is no extensive discussion on how to respond to flaky tests. Given that many of those studies focus on detection, it is suggested that flaky tests are mostly either fixed (e.g., removing the flaky behaviour) or controlled (e.g., customizing the test so it is executed only under certain conditions). Those studies also offer details on how to fix and control flaky tests. On the other hand, grey literature advocates a strong mitigation strategy focusing on detecting and quarantining flaky tests for further investigation. Those articles and posts generally offer little details on fixing flaky tests. They rather focus on high-level quarantine strategies.

Another difference that we observed is how *flakiness* is being defined in different studies — in general, a test is considered flaky if it has a different outcome on different runs with the same input data. Academic literature refers to tests having binary outcomes, i.e., *PASS* or *FAIL*. In practice, however, tests can have multiple outcomes on execution (pass, fail, error or skip). For instance, tests may be skipped/ignored (potentially non-deterministically) or may not terminate (or timeout, depending on the configuration of tests or test runners). A more concise and consistent definition of the different variants of flakiness is needed.

5.5. Implications on research and practice

This study yields some actionable insights and opportunities for future research. We discuss those implications in the following:

1. The review clearly demonstrates that academic research on test flakiness focuses mostly on Java, with limited studies done in other popular languages[15] i.e., JavaScript and Python. The likely reasons are a combination of existing expertise in this area, the availability of open-source datasets, and the availability of high-quality and low-cost (often free) program analysis tools. For example, one of the most active groups that have extensively published work on test flakiness is Marinov et al. group at the University of Illinois at Urbana-Champaign. The group's work in this area is almost exclusively in Java (e.g., Gyori et al. (2015) and Lam et al. (2020c)). This resulted in Java tools and datasets that were widely used in subsequent studies. For example, DeFlaker (Bell et al., 2018), iFixFlakies (Shi et al., 2019), and iDFlakies (Lam et al., 2019b) datasets (all in Java) have been widely used to evaluate techniques that detect order-dependent tests.
2. In contrast, our grey literature review shows that the focus among practitioners is more on the "big picture", and flakiness has been discussed in the context of a variety of programming languages (and mostly using language-agnostic approaches).
3. Different programming languages have different features, and it is not obvious how results observed in Java programs carry over to other languages. For instance: flakiness caused by test order dependencies and shared (memory) state are not possible in a pure functional language (like Haskell), and at least less likely in a language that manages memory more actively to restrict aliasing (like Rust using ownership[16]). In languages with different concurrency models, such as single-threaded languages (e.g., JavaScript), some flakiness caused by concurrency is less likely to occur. For instance, deadlocks are more common in multithreaded applications (Wang et al., 2017). Still, this does not mean that flakiness cannot occur due to concurrency, but it is likely to happen to a lesser extent compared to multithreaded languages such as Java. Similarly, languages (like Java) that use a virtual machine decoupling the runtime from operating systems and hardware are less likely to produce flakiness due to variability in those platforms than low-level languages lacking such a feature, like C. Languages with strong integrated dynamic/meta programming features to facilitate testing like mock testing, which when used may help avoid certain kinds of flakiness, for instance, flakiness caused by network dependencies.
4. There seems to be an imbalance in the way to respond to flaky tests between what has been discussed in academic and industry articles. Industry responses have focused on processes to deal with flaky tests (such as quarantining strategies), and academic research has focused more on causes detection (note that there are some recent studies on automatically repairing flakiness in ML projects and order-dependent tests). This is not unexpected, however, and may also indicate opportunities for future academic research to provide tools that can help automate quarantining (and de-quarantining). Furthermore, it appears that

[15] Based on Stack Overflow language popularity statistics https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies.

[16] https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

some industrial practices, such as rerunning failed tests until they pass, may require a deeper theoretical foundation. For instance, does a test that only passes after several reruns provide the same level of assurance as a test that always passes provides? The same question can be asked for entire test suites: what is the quality of a test suite that never passes in its entirety, but each individual test is observed to pass in some configuration?

5. Another question arises from this, what is the number of reruns required to assure (with a high level of confidence level) that a test is not flaky? From what we observed in the studies that used a rerun approach to manifest flakiness, the number of reruns used differs from one study to another (with some studies noting 2 Dong et al., 2021a, 10 Morán Barbón et al., 2020 or even 100 Lam et al., 2019a reruns as baselines). A recent study on Python projects reported that ∼170 reruns are required to ensure a test is not flaky due to non-order-dependent reasons (Gruber et al., 2021b). We believe that the number of reruns required will depend largely on the cause of flakiness. Some rerun-based tools, such as rspec-retry[17] for Ruby or the `RepeatedTest`[18] annotation in JUnit, provides an option to rerun tests a specified n number of times (set by the developer). However, it is unknown what is a suitable threshold for the number of reruns required for different types of flakiness. A further empirical investigation is required to quantify the minimum number of reruns required to manifest flakiness (for the different causes and contexts). Dong et al. (2021b) is a step in this direction where they rerun tests in 24 Java projects 10,000 times to find out how many flaky tests can be found with different numbers of reruns. Furthermore, using rerun to detect flakiness while simulating nondeterminism is more common in studies that vary test orders to expose dependencies.
6. Reproducibility of studies involving flakiness is also an issue (Pinto et al., 2020; Lam et al., 2020c), which suggests rerun is not the best approach for reproducibility as other confounding factors are not controlled for. From our own experience, it is hard to replicate findings (e.g., reveal the same test flakiness in the test suite) with the same number of reruns in a controlled environment (in our case, to reveal flakiness in the presence of instrumentation Rasheed et al., 2023). This will harden the community effort in reproducing flakiness' research findings. More effort is required to investigate the reproducibility of flaky tests in more depth. FlakiMe (Cordy et al., 2022) is a step in this direction, which is a platform for flakiness experimentation where the degree of flakiness can be controlled.
7. Most empirical studies of flakiness qualitatively examine flakiness fixing commits and then categorize causal factors and strategies for fixing flaky tests. However, this can be problematic given that these commits may not be true fixes as many flaky tests can still be unfixed, which makes this approach unreliable in revealing root causes and mitigation strategies (Lam et al., 2020a). We note only one study (Gruber et al., 2021a) that actually reruns tests and objectively measures flakiness by examining test outcomes over multiple reruns (rather than finding fix commits).

[17] https://github.com/NoRedInk/rspec-retry.

[18] https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/RepeatedTest.html.

6. Validity threats

We discuss a number of potential threats to the validity of the study below, and explain the steps taken to mitigate them.

Incomplete or inappropriate selection of articles: As with any systematic review study, due to the use of an automatic search it is possible that we may have missed some articles that were not either covered by our search string or were not captured by our search tool. We mitigated this threat by first running and refining our search string multiple times. We piloted the search string on Google Scholar to check what the string will return. We cross-validated this by checking if the search string would return well-known, highly cited articles of test flakiness (e.g., Luo et al. (2022), Memon and Cohen (2013) and Eck et al. (2019b)). We believe this iterative approach has improved our search string and reduced the risk of missing key articles.

There is also a chance that some related articles have used terms other than those we used in our search string. If terms other than "flaky", "flakiness" or "non-deterministic" were used, then the possibility of missing those studies increases. To avoid such a limitation we repeatedly refined our search string and performed sequential testing in order to recognize and include as many relevant studies as possible.

Manual analysis of articles: We read through each of the academic and grey literature articles in order to answer our research questions. This was done manually with at least one of the authors reading through articles and then the overall results are verified by another co-author. This manual analysis could introduce bias due to multiple interpretations and/or oversight. We are aware that human interpretation introduces bias, and thus we attempted to account for it via cross-validation involving multiple evaluators and by cross-checking the results from the classification stage, by involving at least two coders.

Classification and reliability: We have performed a number of classifications based on findings from different academic and grey literature articles to answer our four research questions. We extracted information such as causes of flakiness (RQ1), detection methods and tools (RQ2), the impact of flakiness (RQ3) and responses (RQ4). This information was obtained by reading through the articles, extracting the relevant information, and then classifying the articles by one of the authors. Another author then cross-validated the overall classification of articles. We made sure that at least two of the co-authors will check each result and discuss any difference until a 100% agreement between the two is reached.

7. Conclusion

This paper systematically studied how test flakiness has been addressed in academic and grey literature. We provide a comprehensive view of flaky tests, their common causes, their impact on other techniques/artefacts and discuss response strategies in research and practice. We also studied methods and tools that have been used to detect and locate flaky tests and strategies followed in responding to flaky tests.

This review covers 200 articles/posts (including 109 academic articles that primarily focus on test flakiness and 91 grey literature articles). The results show that most academic studies covering test flakiness have focused more on Java than other programming languages. Regarding the common causes, we observed that flakiness due to test order dependency and concurrency had been studied more widely compared to other noted sources of flakiness. However, this depends mainly on the focus of the studies that reported those causes. For example, studies that used Android as their subject systems have focused mostly on flakiness in UI (to which concurrency issues are attributed as the root cause). Correspondingly, methods to detect flaky tests have focused more on specific types of flaky tests, with the dynamic rerun-based approach noted as the main proposed method for flaky test detection. The intention is to provide approaches (either static or dynamic) that are less expensive to run by accelerating ways to manifest flakiness by running fewer tests.

This paper outlines some limitations in test flakiness research that should be addressed by researchers in the future.

CRediT authorship contribution statement

Amjed Tahir: Conceptualization, Structuring, Literature review, Methodology, Results, Discussion, Writing – review & editing. **Shawn Rasheed:** Conceptualization, Structuring, Literature review, Methodology, Results, Discussion, Writing – review & editing. **Jens Dietrich:** Methodology, Discussion, Review & editing. **Negar Hashemi:** Literature review, Data validation, Review. **Lu Zhang:** Discussion, Review & editing.

Data availability

No data was used for the research described in the article.

Acknowledgements

This work is funded by Science for Technological Innovation (SfTI) National Science Challenge (NSC) of New Zealand, grant number MAUX2004.

References

Adams, J., Hillier-Brown, F.C., Moore, H.J., Lake, A.A., Araujo-Soares, V., White, M., Summerbell, C., 2016. Searching and synthesising 'grey literature'and 'grey information'in public health: critical reflections on three case studies. Syst. Rev. 5 (1), 1–11.

Ahmad, A., Leifler, O., Sandahl, K., 2021. Empirical analysis of practitioners' perceptions of test flakiness factors. Softw. Test. Verif. Reliab. 31 (8), e1791.

Barboni, M., Bertolino, A., Angelis, G.D., 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.

Butijn, B.-J., Tamburri, D.A., van den Heuvel, W.-J., 2020. Blockchains: A systematic multivocal literature review. ACM Comput. Surv. 53 (3), 1–37.

Dietrich, J., Rasheed, S., Tahir, A., 2022. Flaky test sanitisation via on-the-fly assumption inference for tests with network dependencies. In: 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE.

Dong, Z., Tiwari, A., Yu, X.L., Roychoudhury, A., 2021c. 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.

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

Ernst, M.D., 2003. Static and dynamic analysis: Synergy and duality. In: IN WODA 2003: ICSE Workshop on Dynamic Analysis.

Fowler, M., 2011b. Eradicating non-determinism in tests.

Fraser, G., Arcuri, A., 2011. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419.

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

Garousi, V., Felderer, M., Mäntylä, M.V., 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. In: EASE '16, Association for Computing Machinery, New York, NY, USA, pp. 1–6, no. Article 26.

Garousi, V., Felderer, M., Mäntylä, M.V., 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf. Softw. Technol. 106, 101–121.

Garousi, V., Küçük, B., 2018. Smells in software test code: A survey of knowledge in industry and academia. J. Syst. Softw. 138, 52–81.

Garousi, V., Mäntylä, M.V., 2016. When and what to automate in software testing? A multi-vocal literature review. Inf. Softw. Technol. 76, 92–117.

Glass, R.L., DeMarco, T., 2006. Software Creativity 2.0. developer.* Books.

Gruber, M., Lukasczyk, S., Kroiß, F., Fraser, G., 2021b. An empirical study of flaky tests in python. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 148–158.

Habchi, S., Haben, G., Papadakis, M., Cordy, M., Le Traon, Y., 2022. 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.

Harman, M., O'Hearn, P., 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 1–23.

Hashemi, N., Tahir, A., Rasheed, S., 2022. An empirical study of flaky tests in JavaScript. In: 2022 38th IEEE International Conference on Software Maintenance and Evolution (ICSME).

Islam, C., Babar, M.A., Nepal, S., 2019. A Multi-Vocal review of security orchestration. ACM Comput. Surv. 52 (2), 1–45.

Kitchenham, B., Charters, S., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. EBSE Technical Report, EBSE-2007-01.

Labuschagne, A., Inozemtseva, L., Holmes, R., 2017. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, pp. 821–830, [Online]. Available: https://doi.org/10.1145/3106237.3106288.

Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2022. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 643–653.

Mahood, Q., Van Eerd, D., Irvin, E., 2014. Searching for grey literature for systematic reviews: challenges and benefits. Res. Synth. Methods 5 (3), 221–234.

McGuinness, D.L., Van Harmelen, F., et al., 2004. OWL web ontology language overview. W3C Recomm. 10 (10), 2004.

Memon, A.M., Cohen, M.B., 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 1479–1480.

Myrbakken, H., Colomo-Palacios, R., 2017. DevSecOps: a multivocal literature review. In: International Conference on Software Process Improvement and Capability Determination. Springer, pp. 17–29.

Neuhaus, C., Neuhaus, E., Asher, A., Wrede, C., 2006. The depth and breadth of google scholar: An empirical study. Port.: Libr. Acad. 6 (2), 127–141.

Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2021. A survey of flaky tests. ACM Trans. Softw. Eng. Methodol. 31 (1), 1–74.

Pereira-Vale, A., Fernandez, E.B., Monge, R., Astudillo, H., Márquez, G., 2021. Security in microservice-based systems: A multivocal literature review. Comput. Secur. 102200.

Raine, J., 2020. Reducing flaky builds by 18x. [Online]. Available: https://github.blog/2020-12-16-reducing-flaky-builds-by-18x/.

Rasheed, S., Dietrich, J., Tahir, A., 2023. On the effect of instrumentation on test flakiness. In: 4th ACM/IEEE International Conference on Automation of Software Test (AST). IEEE.

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.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516.

Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F., 2003. Model checking programs. Autom. Softw. Eng. 10 (2), 203–232.

Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., Wei, J., 2017. A comprehensive study on real world concurrency bugs in node. js. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 520–531.

Williams, A., 2019. Finding High-Quality Grey Literature for Use as Evidence in Software Engineering Research Ph.D. dissertation. University of Canterbury.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.

Yasin, A., Fatima, R., Wen, L., Afzal, W., Azhar, M., Torkar, R., 2020. On using grey literature and google scholar in systematic literature reviews in software engineering. IEEE Access 8, 36226–36243.

Zheng, W., Liu, G., Zhang, M., Chen, X., Zhao, W., 2021. Research progress of flaky tests. In: 3rd International Workshop on Intelligent Bug Fixing. IEEE, pp. 639–646.

Zolfaghari, B., Parizi, R.M., Srivastava, G., Hailemariam, Y., 2020. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. Softw. - Pract. Exp..

References for Academic Literature

Abdi, M., Rocha, H., Demeyer, S., Bergel, A., 2021. Small-amp: Test amplification in a dynamically typed language. [Online]. Available: https://arxiv.org/abs/2108.05663.

Ahlgren, J., Berezin, M., Bojarczuk, K., Dulskyte, E., Dvortsova, I., George, J., Gucevska, N., Harman, M., Lomeli, M., Meijer, E., Sapora, S., Spahr-Summers, J., 2021. Testing web enabled simulation at scale using metamorphic testing. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 140–149.

Ahmad, A., Leifler, O., Sandahl, K., 2020. An evaluation of machine learning methods for predicting flaky tests. In: QuASoQ@APSEC.

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). pp. 1572–1584.

Alvaro, P., Hutchinson, A., Conway, N., Marczak, W.R., Hellerstein, J.M., 2012. BloomUnit: Declarative testing for distributed programs. In: Proceedings of the Fifth International Workshop on Testing Database Systems. DBTest '12, Association for Computing Machinery, New York, NY, USA, [Online]. Available: https://doi.org/10.1145/2304510.2304512.

Bell, J., Kaiser, G., 2014. Unit test virtualization with VMVM. In: Proceedings of the 36th International Conference on Software Engineering. pp. 550–561.

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.

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.

Beller, M., Wong, C.-P., Bader, J., Scott, A., Machalica, M., Chandra, S., Meijer, E., 2021. What it would take to use mutation testing in industry—A study at facebook. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 268–277.

Berglund, A., Vateman, O., 2020. Mitigation and handling of non-deterministic tests in automatic regression testing. In: LU-CS-EX.

Biagiola, M., Stocco, A., Mesbah, A., Ricca, F., Tonella, P., 2019. Web test dependency detection. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 154–164, [Online]. Available: https://doi.org/10.1145/3338906.3338948.

Böhme, M., 2019. Assurances in software testing: A roadmap. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, pp. 5–8.

Camara, B., Silva, M., Endo, A., Vergilio, S., 2021a. On the use of test smells for prediction of flaky tests. In: Brazilian Symposium on Systematic and Automated Software Testing. Association for Computing Machinery, New York, NY, USA, pp. 46–54, [Online]. Available: https://doi.org/10.1145/3482909.3482916.

Camara, B.H.P., Silva, M.A.G., Endo, A.T., Vergilio, S.R., 2021b. What is the vocabulary of flaky tests? An extended replication. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). pp. 444–454.

Candido, J., Melo, L., d'Amorim, M., 2017. Test suite parallelization in open-source projects: a study on its usage and impact. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 838–848.

Chan, W., Cheng, M., Cheung, S., Tse, T., 2007. Automated goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: An experimental case study. Qual. Control Appl. Statist. 52 (1), 113–114.

Chen, J., Shang, W., Shihab, E., 2020. PerfJIT: Test-level just-in-time prediction for performance regression introducing commits. IEEE Trans. Softw. Eng..

Cox, S., Chen, N., 2019. Improving client side web testing automation in continuous integration-a case study. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP). The Steering Committee of The World Congress in Computer Science, Computer …, pp. 41–47.

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.

Danglot, B., Vera-Pérez, O.L., Baudry, B., Monperrus, M., 2019. Automatic test improvement with DSpot: a study with ten mature open-source projects. Empir. Softw. Eng. 24 (4), 2603–2635.

Demeyer, S., Parsai, A., Vercammen, S., van Bladel, B., Abdi, M., 2020. Formal verification of developer tests: a research agenda inspired by mutation testing. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II 9. Springer, pp. 9–24.

de Oliveira Neto, F.G., Dobslaw, F., Feldt, R., 2020. Using mutation testing to measure behavioural test diversity. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 254–263.

Dong, Z., Tiwari, A., Yu, X.L., Roychoudhury, A., 2021a. Concurrency-related flaky test detection in android apps.

Dong, Z., Tiwari, A., Yu, X.L., Roychoudhury, A., 2021b. 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. In: ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, pp. 367–378, [Online]. Available: https://doi.org/10.1145/3468264.3468584.

Dorward, B., Johnston, C., Nickell, E., Henderson, T., 2021. Flake-aware culprit finding.

Durieux, T., Le Goues, C., Hilton, M., Abreu, R., 2020. Empirical study of restarted and flaky builds on travis CI. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR '20, Association for Computing Machinery, New York, NY, USA, pp. 254–264, [Online]. Available: https://doi.org/10.1145/3379597.3387460.

Dutta, S., Arunachalam, A., Misailovic, S., 2022. To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects. In: 15th IEEE International Conference on Software Testing, Verification and Validation.

Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., Misailovic, S., 2020. Detecting flaky tests in probabilistic and machine learning applications. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, pp. 211–224.

Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A., 2019a. Understanding flaky tests: The developer's perspective. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 830–840, [Online]. Available: https://doi.org/10.1145/3338906.3338945.

Eddins, S.L., 2009. Automated software testing for matlab. Comput. Sci. Eng. 11 (6), 48–55.

Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2014, Association for Computing Machinery, New York, NY, USA, pp. 235–245, [Online]. Available: https://doi.org/10.1145/2635868.2635910.

Eloussi, L., 2015. Determining flaky tests from test failures.

Endo, A.T., Møller, A., 2020. NodeRacer: Event race detection for node.js applications. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 120–130.

Erlenhov, L., de Oliveira Neto, F.G., Chukaleski, M., Daknache, S., 2020. Challenges and guidelines on designing test cases for test bots. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSEW '20, Association for Computing Machinery, New York, NY, USA, pp. 41–45, [Online]. Available: https://doi.org/10.1145/3387940.3391535.

Fan, Z., 2019. A systematic evaluation of problematic tests generated by EvoSuite. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 165–167.

Fatima, S., Ghaleb, T.A., Briand, L., 2021. Flakify: A black-box, language model-based predictor for flaky tests. [Online]. Available: https://arxiv.org/abs/2112.12331.

Gabrielova, E., 2017. End-to-end regression testing for distributed systems. In: Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference. Middleware '17, Association for Computing Machinery, New York, NY, USA, pp. 9–12, [Online]. Available: https://doi.org/10.1145/3152688.3152692.

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

Gay, G., 2017. Generating effective test suites by combining coverage criteria. In: International Symposium on Search Based Software Engineering. Springer, pp. 65–82.

Ginelli, D., Martinez, M., Mariani, L., Monperrus, M., 2020. A comprehensive study of code-removal patches in automated program repair. arXiv preprint arXiv:2012.06264.

Grano, G., Palomba, F., Di Nucci, D., De Lucia, A., Gall, H.C., 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. J. Syst. Softw. 156, 312–327.

Gruber, M., Fraser, G., 2022. A survey on how test flakiness affects developers and what support they need to address it. arXiv preprint arXiv:2203.00483.

Gruber, M., Lukasczyk, S., Kroiß, F., Fraser, G., 2021a. An empirical study of flaky tests in python. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 148–158.

Gyori, A., Lambeth, B., Khurshid, S., Marinov, D., 2017. Exploring underdetermined specifications using java PathFinder. SIGSOFT Softw. Eng. Not. 41 (6), 1–5, [Online]. Available: https://doi.org/10.1145/3011286.3011295.

Gyori, A., Lambeth, B., Shi, A., Legunsen, O., Marinov, D., 2016. NonDex: A tool for detecting and debugging wrong assumptions on java API specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 993–997, [Online]. Available: https://doi.org/10.1145/2950290.2983932.

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

Haas, R., Elsner, D., Juergens, E., Pretschner, A., Apel, S., 2021. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 1281–1291, [Online]. Available: https://doi.org/10.1145/3468264.3473922.

Habchi, S., Haben, G., Papadakis, M., Cordy, M., Traon, Y.L., 2021. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. arXiv preprint arXiv:2112.04919.

Haben, G., Habchi, S., Papadakis, M., Cordy, M., Le Traon, Y., 2021a. 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). pp. 219–229.

Haben, G., Habchi, S., Papadakis, M., Cordy, M., Traon, Y.L., 2021b. Discerning legitimate failures from false alerts: A study of chromium's continuous integration.

Herzig, K., Nagappan, N., 2015. Empirically detecting false test alarms using association rules. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. pp. 39–48.

Hirsch, T., Schindler, C., Müller, M., Schranz, T., Slany, W., 2019. An approach to test classification in big android applications. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, pp. 300–308.

Ivanković, M., Petrović, G., Just, R., Fraser, G., 2019. Code coverage at google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 955–963, [Online]. Available: https://doi.org/10.1145/3338906.3340459.

Jeon, J., Hong, S., 2020. Threats to validity in experimenting mutation-based fault localization. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results. In: ICSE-NIER '20, Association for Computing Machinery, New York, NY, USA, pp. 1–4, [Online]. Available: https://doi.org/10.1145/3377816.3381746.

King, T.M., Santiago, D., Phillips, J., Clarke, P.J., 2018. Towards a Bayesian network model for predicting flaky automated tests. In: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 100–107.

Koivuniemi, J., 2017. Shortening feedback time in continuous integration environment in large-scale embedded software development with test selection. Univ. Oulu Repos. 16–18.

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. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 101–111, [Online]. Available: https://doi.org/10.1145/3293882.3330570.

Lam, W., Muşlu, K., Sajnani, H., Thummalapenta, S., 2020a. A study on the lifecycle of flaky tests. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 1471–1482, [Online]. Available: https://doi.org/10.1145/3377811.3381749.

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). pp. 312–322.

Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M.D., Xie, T., 2020b. Dependent-test-aware regression testing techniques. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 298–311, [Online]. Available: https://doi.org/10.1145/3395363.3397364.

Lam, W., Winter, S., Astorga, A., Stodden, V., Marinov, D., 2020c. 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.

Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., Bell, J., 2020d. A large-scale longitudinal study of flaky tests. Proc. ACM Program. Lang. 4, [Online]. Available: https://doi.org/10.1145/3428270.

Lassila, A., et al., 2019. Opportunities and challenges in adopting continuous end-to-end testing: A case study.

Laurent, T., Wall, F., Ventresque, A., 2020. On the impact of timeouts and JVM crashes in pitest. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 247–253.

Li, C., Zhu, C., Wang, W., Shi, A., 2022. Repairing Order-Dependent Flaky Tests Via Test Generation. ICSE.

Linares-Vásquez, M., Moran, K., Poshyvanyk, D., 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 399–410.

Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 643–653.

Machalica, M., Samylkin, A., Porth, M., Chandra, S., 2019. Predictive test selection. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 91–100.

MacIver, D.R., Donaldson, A.F., 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In: Hirschfeld, R., Pape, T. (Eds.), 34th European Conference on Object-Oriented Programming (ECOOP 2020). In: Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 13:1–13:27, [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13170.

Malm, J., Causevic, A., Lisper, B., Eldh, S., 2020. Automated analysis of flakiness-mitigating delays. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test. AST '20, Association for Computing Machinery, New York, NY, USA, pp. 81–84, [Online]. Available: https://doi.org/10.1145/3387903.3389320.

Mansky, S., Gunter, E.L., 2020. Safety of a smart classes-used regression test selection algorithm. Electron. Notes Theor. Comput. Sci. 351, 51–73.

Mårtensson, T., Ståhl, D., Bosch, J., 2016. Continuous integration applied to software-intensive embedded systems – problems and experiences. In: Abrahamsson, P., Jedlitschka, A., Nguyen Duc, A., Felderer, M., Amasaki, S., Mikkonen, T. (Eds.), Product-Focused Software Process Improvement. Springer International Publishing, Cham, pp. 448–457.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 22 (4), 1936–1964.

Mascheroni, M.A., Irrazábal, E., 2018. Identifying key success factors in stopping flaky tests in automated REST service testing. J. Comput. Sci. Tech. 18 (02), e16.

Mascheroni, M.A., Irrazábal, E., Carruthers, J.A., Pinto, J.A., 2019. Rapid releases and testing problems at the industry: A survey. In: XXV Congreso Argentino de Ciencias de la Computación (CACIC)(Universidad Nacional de Río Cuarto, Córdoba, 14 al 18 de octubre de 2019).

Mascheroni, M.A., Irrazábal, E., Rossi, G., 2021. Continuous testing improvement model. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST). pp. 109–112.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J., 2017. Taming google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, pp. 233–242.

Mendes, D.J.G., 2019. Automated Testing for Provisioning Systems of Complex Cloud Products Ph.D. dissertation.

Mondal, S., Silva, D., d'Amorim, M., 2021. Soundy automated parallelization of test execution. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 309–319.

Morán Barbón, J., Augusto Alonso, C., Bertolino, A., Riva Álvarez, C.A., García Tuya, J.F., et al., 2019. Debugging flaky tests on web applications. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies-Volume 1: APMDWE.

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. J. Web Eng. 2.

Mudduluru, R., Waataja, J., Millstein, S., Ernst, M., 2021. Verifying determinism in sequential programs. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 37–49.

Olianas, D., Leotta, M., Ricca, F., Villa, L., 2021. Reducing flakiness in end-to-end test suites: An experience report. In: International Conference on the Quality of Information and Communications Technology. Springer, pp. 3–17.

Palomba, F., Zaidman, A., 2017. Notice of retraction: Does refactoring of test smells induce fixing flaky tests? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 1–12.

Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2020. Flake it 'till you make it: Using automated repair to induce and fix latent test flakiness. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSEW '20, Association for Computing Machinery, New York, NY, USA, pp. 11–12, [Online]. Available: https://doi.org/10.1145/3387940.3392177.

Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2022. Evaluating features for machine learning detection of order-and non-order-dependent flaky tests. In: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE.

Paydar, S., Azamnouri, A., 2019. An experimental study on flakiness and fragility of randoop regression test suites. In: International Conference on Fundamentals of Software Engineering. Springer, pp. 111–126.

Person, S., Elbaum, S., 2015. Test analysis: Searching for faults in tests (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 149–154.

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.

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. In: MaLTESQuE 2021, Association for Computing Machinery, New York, NY, USA, pp. 19–24, [Online]. Available: https://doi.org/10.1145/3472674.3473981.

Ramler, R., Klammer, C., Wetzlmaier, T., 2019. Lessons learned from making the transition to model-based GUI testing. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. In: A-TEST 2019, Association for Computing Machinery, New York, NY, USA, pp. 22–27, [Online]. Available: https://doi.org/10.1145/3340433.3342823.

Rehman, M.H.U., Rigby, P.C., 2021. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, pp. 1371–1380, [Online]. Available: https://doi.org/10.1145/3468264.3473930.

Schwahn, O., Coppik, N., Winter, S., Suri, N., 2019. Assessing the state and improving the art of parallel testing for c. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 123–133, [Online]. Available: https://doi.org/10.1145/3293882.3330573.

Scott, E.O., Luke, S., 2019. ECJ at 20: Toward a general metaheuristics toolkit. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. GECCO '19, Association for Computing Machinery, New York, NY, USA, pp. 1391–1398, [Online]. Available: https://doi.org/10.1145/3319619.3326865.

Shi, A.W., 2020. Improving Regression Testing Efficiency and Reliability Via Test-Suite Transformations Ph.D. dissertation. University of Illinois at Urbana-Champaign.

Shi, A., Gyori, A., Mahmood, S., Zhao, P., Marinov, D., 2018. Evaluating test-suite reduction in real software evolution. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2018, Association for Computing Machinery, New York, NY, USA, pp. 84–94, [Online]. Available: https://doi.org/10.1145/3213846.3213875.

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

Silva, B.C.F., Carvalho, G., Sampaio, A., 2019. CPN simulation-based test case generation from controlled natural-language requirements. Sci. Comput. Program. 181, 111–139.

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). pp. 301–311.

Soltani, M., Derakhshanfar, P., Devroey, X., Van Deursen, A., 2020. A benchmark-based evaluation of search-based crash reproduction. Empir. Softw. Eng. 25 (1), 96–138.

Storey, M.-A., Zagalsky, A., 2016. Disrupting developer productivity one bot at a time. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 928–931, [Online]. Available: https://doi.org/10.1145/2950290.2983989.

Strandberg, P.E., Ostrand, T.J., Weyuker, E.J., Afzal, W., Sundmark, D., 2020. Intermittently failing tests in the embedded systems domain. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 337–348, [Online]. Available: https://doi.org/10.1145/3395363.3397359.

Terragni, V., Salza, P., Ferrucci, F., 2020. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results. pp. 69–72.

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). pp. 534–538.

Tomassi, D.A., Rubio-González, C., 2019. A note about: Critical review of BugSwarm for fault localization and program repair. arXiv preprint arXiv:1910.13058.

Urli, S., Yu, Z., Seinturier, L., Monperrus, M., 2018. How to design a program repair bot? insights from the repairnator project. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, pp. 95–104.

Vaidhyam Subramanian, S.V., McIntosh, S., Adams, B., 2020. Quantifying, characterizing, and mitigating flakily covered program elements. IEEE Trans. Softw. Eng. 1.

Vehabovic, A., 2020. The process of changing out expandable elements in a large-scale web application.

Vahabzadeh, A., Stocco, A., Mesbah, A., 2018. Fine-grained test minimization. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 210–221.

Vancsics, B., Gergely, T., Beszédes, Á., 2020. Simulating the effect of test flakiness on fault localization effectiveness. In: 2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST). IEEE, pp. 28–35.

Vassallo, C., Proksch, S., Jancso, A., Gall, H.C., Di Penta, M., 2020. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 327–337, [Online]. Available: https://doi.org/10.1145/3368089.3409709.

Verdecchia, R., Cruciani, E., Miranda, B., Bertolino, A., 2021. Know you neighbor: Fast static prediction of test flakiness. IEEE Access.

Wang, R., Chen, Y., Lam, W., 2022. iPFlakies: A framework for detecting and fixing python order-dependent flaky tests.

Wei, A., Yi, P., Li, Z., Xie, T., Marinov, D., Lam, W., 2022. Preempting flaky tests via non-idempotent-outcome tests. In: International Conference on Software Engineering (ICSE'22).

Wei, A., Yi, P., Xie, T., Marinov, D., Lam, W., 2021. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 270–287.

White, M., Tufano, M., Martinez, M., Monperrus, M., Poshyvanyk, D., 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 479–490.

Widder, D.G., Hilton, M., Kästner, C., Vasilescu, B., 2019. A conceptual replication of continuous integration pain points in the context of travis CI. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 647–658, [Online]. Available: https://doi.org/10.1145/3338906.3338922.

Wild, N., Lichter, H., Kehren, P., 2020. Test automation challenges for application landscape frameworks. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 330–333.

Ye, H., Martinez, M., Monperrus, M., 2021. Automated patch assessment for program repair at scale. Empir. Softw. Eng. 26 (2), 1–38.

Yi, P., Wei, A., Lam, W., Xie, T., Marinov, D., 2021. Finding polluter tests using java PathFinder. SIGSOFT Softw. Eng. Not. 46 (3), 37–41, [Online]. Available: https://doi.org/10.1145/3468744.3468756.

Zdun, U., Wittern, E., Leitner, P., 2019. Emerging trends, challenges, and experiences in DevOps and microservice APIs. IEEE Softw. 37 (1), 87–91.

Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D., 2014. Empirically revisiting the test independence assumption. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 385–396.

Zhang, P., Jiang, Y., Wei, A., Stodden, V., Marinov, D., Shi, A., 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 50–61.

Zhu, H., Wei, L., Wen, M., Liu, Y., Cheung, S.-C., Sheng, Q., Zhou, C., 2020. MockSniffer: Characterizing and recommending mocking decisions for unit tests. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 436–447.

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). pp. 736–745.

References for Grey Literature

Colantonio, J., 2015. 10 reasons for flaky tests automation + tips | test guild. [Online]. Available: https://testguild.com/top-10-reasons-for-flaky-automated-tests/.

Snyder, S., 2021. A pragmatist's guide to flaky test management. [Online]. Available: https://gradle.com/blog/a-pragmatists-guide-to-flaky-test-management/.

Agarwal, R., 2021. Handling flaky unit tests in java. [Online]. Available: https://eng.uber.com/handling-flaky-tests-java/.
Andrawis, D., 2020. Tips on how to reduce Selenium flaky tests. [Online]. Available: https://www.shield34.com/blog/specific-tips-on-how-to-reduce-selenium-flaky-tests-that-every-coder-should-know/.
2017. Automated testing - how to deal with flaky tests that have intermittent failures? [Online]. Available: https://sqa.stackexchange.com/questions/28204/how-to-deal-with-flaky-tests-that-have-intermittent-failures.
Berczuk, S., 2020. Defensive design strategies to prevent flaky tests. [Online]. Available: https://www.techwell.com/techwell-insights/2020/04/defensive-design-strategies-prevent-flaky-tests.
Chodorow, K., 2015. Debugging flaky tests with Bazel. [Online]. Available: https://kchodorow.com/2015/09/17/debugging-flaky-tests-with-bazel/.
Datadog, Flaky test management. [Online]. Available: https://docs.datadoghq.com/continuous_integration/guides/flaky_test_management/.
2020. Detect, track and eliminate flaky tests. [Online]. Available: https://buildpulse.io/.
Pierce, E., 2021. eBay launches targeted auto retry. [Online]. Available: https://tech.ebayinc.com/engineering/ebay-launches-targeted-auto-retry/.
Eloussi, L., 2016. Flaky tests (and how to avoid them). [Online]. Available: https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.
Gladhorn , F., 2017. Fixing a flaky test - how hard can it be? [Online]. Available: https://www.qt.io/blog/2017/05/12/fixing-a-flaky-test-how-hard-can-it-be.
Flaky tests — pytest documentation. [Online]. Available: https://docs.pytest.org/en/stable/flaky.html.
Flaky tests | GitLab. [Online]. Available: https://docs.gitlab.com/ee/development/testing_guide/flaky_tests.html.
Schwering, R., 2021. Flaky tests: Getting rid of a living nightmare in testing. [Online]. Available: https://www.smashingmagazine.com/2021/04/flaky-tests-living-nightmare/.
2017. Flaky tests - a war that never ends | hacker noon. [Online]. Available: https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359.
Flaky test extractor Maven plugin. [Online]. Available: https://github.com/zeebe-io/flaky-test-extractor-maven-plugin.
Fowler, M., 2011a. Eradicating non-determinism in tests. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html.
Gradle enterprise flaky test detection guide | gradle enterprise docs. [Online]. Available: https://docs.gradle.com/enterprise/flaky-test-detection/.
Grant, J., 2016. What flaky tests can tell you. [Online]. Available: https://www.stickyminds.com/article/what-flaky-tests-can-tell-you.
Hallam, J., 2019. Avoiding flaky tests. [Online]. Available: https://developers.mattermost.com/blog/avoiding-flaky-tests/.
Hoa, K., 2015. Flaky tests & capybara best practices. [Online]. Available: https://www.simplybusiness.co.uk/about-us/tech/2015/02/flaky-tests-and-capybara-best-practices/.
Machalica, M., Chmiel, W., Swierc, S., Sakevych, S., 2020. How do you test your tests? [Online]. Available: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/.
özal, S., 2021. How to deal with flaky tests. [Online]. Available: https://thenewstack.io/how-to-deal-with-flaky-tests/.
Tagwerker, E., 2019. How to debug non-deterministic test failures with rspec - FastRuby.io | rails upgrade service. [Online]. Available: https://fastruby.io/blog/rspec/debug/how-to-debug-non-deterministic-specs.html.
Powell, R., 2022. How to reduce flaky test failures. [Online]. Available: https://circleci.com/blog/reducing-flaky-test-failures/.
Thummalapenta, S., 2022. Improving developer productivity via flaky test management. [Online]. Available: https://devblogs.microsoft.com/engineering-at-microsoft/improving-developer-productivity-via-flaky-test-management/.
Debregziabher, D., 2021. Introducing test insights with flaky test detection. [Online]. Available: https://circleci.com/blog/introducing-test-insights-with-flaky-test-detection/.
Katalon studio 8.1 – ways to handle flaky tests smarter. [Online]. Available: https://katalon.com/resources-center/blog/studio-8-1-handle-flaky-tests.
Klotz, Andrew, 2019. Imagining better flaky test management. [Online]. Available: https://klotzandrew.com/blog/imagine-better-flaky-tests.
Lapierre, M., 2017. Pros and cons of quarantined tests. [Online]. Available: https://dev.to/mlapierre/pros-and-cons-of-quarantined-tests-2emj.
Lee, B., 2020. We have a flaky test problem. [Online]. Available: https://medium.com/scopedev/how-can-we-peacefully-co-exist-with-flaky-tests-3c8f94fba166.
Listfield, J., 2017. Where do our flaky tests come from? [Online]. Available: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.
2020. Manage flaky tests - azure pipelines. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management.
Managing test flakiness. [Online]. Available: https://smartbear.com/resources/ebooks/managing-ui-test-flakiness/.
Maven Surefire Plugin – Rerun failing tests. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html.
Mayer, D., 2019. Flaky ruby tests. [Online]. Available: https://www.mayerdan.com/ruby/2019/09/07/flaky-ruby-tests.
McCrary, J., 2020. Using Bazel to help fix flaky tests. [Online]. Available: https://jakemccrary.com/blog/2020/06/28/using-bazel-to-help-fix-flaky-tests/.
McPeak, A., 2018. How to fix a flaky selenium suite. [Online]. Available: https://smartbear.com/en/blog/eliminating-flaky-selenium-tests-forever/.
Meadows, J., 2014. Introducing flaky - a nose test plugin for automatically rerunning flaky tests. [Online]. Available: https://blog.box.com/introducing-flaky-a-nose-test-plugin-for-automatically-rerunning-flaky-tests.
Micco, J., 2016. Flaky tests at google and how we mitigate them. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.
Micco, J., 2017. The state of continuous integration testing @google. [Online]. Available: https://research.google/pubs/pub45880/.
Otonelli, M., 2017. 'Flaky' tests: A short story - the lean software boutique. [Online]. Available: https://www.ombulabs.com/blog/rspec/continuous-integration/how-to-track-down-a-flaky-test.html.
Palmer, J., 2019. Test flakiness - methods for identifying and dealing with flaky tests. [Online]. Available: https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.
Peterson, S., 2019. Fixing flaky tests like a detective (RailsConf 2019). [Online]. Available: https://sonja.codes/fixing-flaky-tests-like-a-detective.
Rakiĉ, P., 2017. Flaky tests: are you sure you want to rerun them? [Online]. Available: https://semaphoreci.com/blog/2017/04/20/flaky-tests.html.
Raposa, R., 2020. Flaky test process - test engineering - confluence. [Online]. Available: https://openedx.atlassian.net/wiki/spaces/TE/pages/161427235/Flaky+Test+Process.
Raine, J., 2020. Reducing flaky builds by 18x. [Online]. Available: https://github.blog/2020-12-16-reducing-flaky-builds-by-18x/.
Richter, L., 2020. Handle flaky tests with quarantine and Xunit.SkippableFact. [Online]. Available: https://dev.to/n_develop/handle-flaky-tests-with-quarantine-and-xunit-skippablefact-3a14.
Rushakoff, M., 2019. Reproducing a flaky test in go | blog. [Online]. Available: https://www.influxdata.com/blog/reproducing-a-flaky-test-in-go/.
Rustamzadeh, A., 2020. Introducing flaky test detection & alerts. [Online]. Available: https://www.cypress.io/blog/2020/10/20/introducing-flaky-test-detection-alerts/.
Saffron, S., 2019. Tests that sometimes fail. [Online]. Available: https://samsaffron.com/archive/2019/05/15/tests-that-sometimes-fail.
SamGu, 2020. Eliminating flaky tests - azure DevOps. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/eliminating-flaky-tests.
Sandhu, A., 2015. How to fix flaky tests. [Online]. Available: https://tech.justeattakeaway.com/2015/03/30/how-to-fix-flaky-tests/.
Shah, U., 2019. Athena: Our automated build health management system. [Online]. Available: https://dropbox.tech/infrastructure/athena-our-automated-build-health-management-system.
Shatrov, K., 2016. Five ways to write a flaky test. [Online]. Available: http://kirshatrov.com/2016/10/21/flaky-tests/.
Stosik, D., 2020. A methodological approach to fixing flaky tests. [Online]. Available: https://sourcediving.com/a-methodological-approach-to-fixing-flaky-tests-92a39162b769.
2019. Strategies for handling flaky test suites - Redshiftzero. [Online]. Available: https://www.redshiftzero.com/test-flakes/.
Striĉeviĉ, N., 2015. How to deal with and eliminate flaky tests. [Online]. Available: https://semaphoreci.com/community/tutorials/how-to-deal-with-and-eliminate-flaky-tests.
Sudarshan, P., 2012. No more flaky tests on the Go team. [Online]. Available: https://qa.webteam.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.
Pirocanac, G., 2020. Test flakiness - one of the main challenges of automated testing. [Online]. Available: https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html.
Testing in Chromium - Fixing web test flakiness. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/testing/identifying_tests_that_depend_on_order.md.
Testinium, 2018. Flaky tests and how to reduce them. [Online]. Available: https://testinium.com/blog/flaky-tests-and-how-to-reduce-them/.
Jones, R., 2021. Top reasons & solutions for test flakiness. [Online]. Available: https://blog.testproject.io/2021/08/25/top-reasons-and-solutions-for-test-flakiness/.
Keča, N., 2017. Tips on treating flakiness in your rails test suite. [Online]. Available: https://semaphoreci.com/blog/2017/08/03/tips-on-treating-flakiness-in-your-test-suite.html.
Tse, I., 2016. How I tracked down a flaky test. [Online]. Available: https://www.paperlesspost.com/blog/teams/how-i-tracked-down-a-flaky-test/.
Wendelin, E., 2019. Identifying and analyzing flaky tests in Maven and Gradle builds. [Online]. Available: https://gradle.com/blog/flaky-tests/.
Yarn, J., 2016. Flaky tests: the tester's f word. [Online]. Available: https://www.lucidchart.com/techblog/2016/12/28/flaky-the-testers-f-word/.

Zhu, E., Solving flaky tests in rspec. [Online]. Available: https://flexport.engineering/solving-flaky-tests-in-rspec-9ceadedeaf0e.

Amjed Tahir is a Senior Lecturer in Software Engineering at Massey University, New Zealand. He obtained a Ph.D. degree from the University of Otago in 2016. His research is mainly in empirical software engineering, with a special interest in software maintenance and testing problems. He has served in the organizing and program committee of several software engineering conferences, including ICSE, ASE and ICSME, and regularly review for leading software engineering journals, including TSE, JSS and EMSE. Amjed is an Executive member of Software Innovation New Zealand (SIÑZ).

Shawn Rasheed is a Lecturer in ICT at Te Pūkenga - Universal College of Learning in Palmerston North, New Zealand. His current research is in using program analysis for software quality,

Jens Dietrich is an Associate Professor in the School of Engineering and Computer Science, at Victoria University of Wellington. He has a Master's in Mathematics and a Ph.D. in Computer Science both from the University of Leipzig. He worked in industry as a consultant in Germany, Namibia, Switzerland and the UK between 1996 and 2003 when he returned to academia. His research interests are in the areas of software componentry and composition and program verification.

Negar is a Ph.D. candidate at Massey University, New Zealand. She is interested in Software Engineering but with a particular focus on software testing, software maintenance, and other software quality topics.

Lu Zhang Is a professor at the School of Computer Science, Peking University, P.R.China. He received both Ph.D. and B.Sc. in Computer Science from Peking University in 2000 and 1995, respectively. He was a post-doctoral researcher at Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM 2008 and a program co-chair of ICSME 2017. He has been on the editorial boards of the Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse and component-based software development, and service computing.