Test Flakiness Across Programming Languages

Keila Barbosa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda

Abstract—Regression Testing (RT) is a quality-assurance practice commonly adopted in the software industry to check if functionality remains intact after code changes. Test flakiness is a serious problem for RT. A test is said to be flaky when it non-deterministically passes or fails on a fixed environment. Prior work studied test flakiness primarily on Java programs. It is unclear, however, how problematic is test flakiness for software written in other programming languages. This paper reports on a study focusing on three central aspects of test flakiness: concentration, similarity, and cost. Considering concentration, our results show that, for any given programming language that we studied (C, Go, Java, JS, and Python), most issues could be explained by a small fraction of root causes (5/13 root causes cover 78.07% of the issues) and could be fixed by a relatively small fraction of fix strategies (10/23 fix strategies cover 85.20% of the issues). Considering similarity, although there were commonalities in root causes and fixes across languages (e.g., concurrency and async wait are common causes of flakiness in most languages), we also found important differences (e.g., flakiness due to improper release of resources are more common in C), suggesting that there is opportunity to fine tuning analysis tools. Considering cost, we found that issues related to flaky tests are resolved either very early once they are posted (<10 days), suggesting relevance, or very late (>100 days), suggesting irrelevance.

Index Terms—Regression Testing, Test Flakiness, Programming Languages

1 Introduction

Regression testing is a very important and common practice in software development today. It is used to check if a functionality implemented in the software remains intact during software evolution. One serious problem that hinders the effectiveness of regression testing is the presence of *flaky tests*. A test is said to be flaky when it non-deterministically passes or fails on the same configuration of the running environment (e.g., code, OS, etc.) [1]. For example, a test could non-deterministically fail because it tries to access a local server –that the test itself spawned– before the server is ready to accept requests. Flaky tests can negatively impact the software development process in two important ways: (1) **Increased Cost**. In the presence of flaky tests, developers cannot decide whether a failure is an indication of a real problem or a false alarm. Developers have to debug the problem, which is time-consuming; (2) **Distrust**. Flaky tests can negatively impact the confidence of organizations in Software Testing; developers may start ignoring the test results, weakening the testing culture [2].

Test flakiness is a serious problem in industry. Many test failures at Google are due to flaky tests [1], [3]. At Microsoft, the presence of flaky tests imposes an important burden on developers. For example, Lam et al. [2] reported that 58 Microsoft developers involved in a survey considered flaky tests to be the second most important reason, out of 10 reasons, for slowing down software deployments. Furthermore, Herzig et al. [4] reported that, at Microsoft, the cost of inspecting test failures due to flakiness is estimated in $7.2 million only considering one product, Microsoft Dynamics. On GitHub, it was reported that "1 in 11 commits had at least one red build caused by a flaky test" [5]. Other organizations share similar problems [6], [7].

Three main strategies exist to deal with flaky tests: (i) prevent, (ii) detect (before running), and (iii) rerun (after failure). Prevention consists of regulating software development to avoid flakiness. For example, at Google, developers are encouraged to write single-threaded tests to avoid flakiness [8]. *Prevention can be challenging*, especially when developers need to write tests other than unit tests. Detection consists of analyzing the test cases before they are executed in regression testing. This strategy has been under active investigation in research. However, *existing detection techniques are limited*. Static detectors are imprecise [9]–[12] and dynamic detectors are limited in scope [13]–[16]. Finally, Rerun consists of re-executing for multiple times test cases that have failed during regression testing. A test that failed and then passed —on a fixed version of the application code— is considered flaky, and vice-versa. The status of a test that persistently failed is unknown, but developers typically treat this scenario as a problem in application code as opposed to a bug in test code. Although Rerun is popular in industry [1], [17], *it is expensive and counter-productive*. Rerunning failing tests consumes a lot of computing power. Google, for example, uses 2-16% of its testing budget just to rerun flaky tests [1], [18]. Rerun is also counter-productive. When developers observe flaky tests during regression testing, they can choose to interrupt their activities to immediately address test flakiness or to provisionally ignore the test, postponing its repair. The decision to interrupt the current task is disruptive whereas the decision to ignore flaky tests, albeit common, is ineffective [19].[1] That practice can result in observations of even more failures during software evolution, as highlighted by Rahman and Rigby [20]. That practice can also reduce the ability of the test suites to detect

- *Keila Costa, Marcelo d'Amorim, and Breno Miranda are with the Federal University of Pernambuco, Brazil.*
  E-mails: {kbcs2,damorim,bafm}@cin.ufpe.br
- *Ronivaldo Ferreira and Gustavo Pinto are with the Federal University of Pará, Brazil*
  E-mails: ronivaldo.junior@icen.ufpa.br and gpinto@ufpa.br

1. For reference, the JUnit annotation @Ignore tells JUnit to ignore the corresponding test.

bugs as it is unclear when developers would eventually revise the ignored test case to eradicate its non-determinism.

This paper uses the term *root cause* (or *source*) to indicate the reason of flakiness and the term *fix strategy* (or *repair*) to refer to the mechanism used by developers to eliminate test flakiness. A root cause can have different fix strategies, and a fix strategy can be used to address different causes. For illustration, Table 1 shows a list of sources of test flakiness, as categorized by Luo et al. [21], whereas Table 3 shows a list of pairs of root causes and fix strategies, as identified by Luo et al. [21] and by Eck et al. [22].

The vast majority of prior work on test flakiness focused on Java (see Section 8). Little is known about how test flakiness manifests in other programming languages. Intuitively, the programming abstractions offered by a PL can influence the prevalence of certain problems and how they are treated. For example, prior work observed that concurrency often induce test flakiness in Java projects (see categories "Async Wait" and "Concurrency" on Table 1). This observation raises the question: *Does concurrency also induce flakiness in projects written in languages other than Java? Are concerns other than concurrency associated with flakiness in projects written in other languages?* The goal of this paper is to understand the root causes and their fixes in popular programming languages. This problem is important for developers writing code in a given language and for developers of (flakiness) analysis tools. For example, machine learning models proposed in the literature to statically classify flaky tests [9]–[12], [23] could be improved based on the understanding that code written in certain PLs are more prone to certain sources of flakiness. Likewise, code repair tools could provide recommendations more likely to be accepted if it is known that certain fixes are more common to address a given root cause in projects written in a given PL.

This paper reports on a large-scale study about flaky tests in projects written in five PLs: C, Go, Java, JavaScript, and Python. The choice of language was primarily based on popularity [24]. Section 3 elaborates on our selection criteria. We mined issues from GitHub using keywords related to test flakiness and inspected those issues to categorize root cause and fix strategy. We stopped the inspection when, for a given language, we confidently categorized the root cause and fix strategy of a total of 100 issues. Each issue was analyzed by two researchers who read the discussion thread, the original code, and the modified code. We used the taxonomies presented by Luo et al. [21] and by Eck et al. [22] to determine the root causes and corresponding strategies to fix flaky tests(see Section 2). Based on the collected data, we posed the following research questions:

RQ1 [Concentration]. *Is it the case that, for a given PL, a small fraction of sources and corresponding fix strategies are associated with the majority of the issues?* This question helps understanding whether it is possible for developers (and tools) to focus on a small set of root causes and fix strategies when analyzing flaky tests. Our results indicate that the answer to this question is affirmative. For example, we found that (1) 5 of the 13 root causes cover 78.07% of the issues and (2) 10 of the 23 fix strategies cover 85.20% of the issues. Curiously, we also found that these proportions are not uniform across languages. For example, the concentration of root causes and fixes in Go was much higher compared to other languages. Three root causes –async wait, concurrency, and network– explain 68.89% of the issues we analyzed on that language.

RQ2 [Similarity]. *How (dis)similar are the root causes and fix strategies across PLs?* The answer to the first question indicates that there is opportunity to improve the analysis of flakiness by focusing on a small portion of root causes and fix strategies. However, it does not explain if the same root causes (and fixes) affect all PLs. Understanding this is important to determine whether specializing analysis tools to specific languages would be helpful. Results to this RQ indicate that there are commonalities and differences (of causes and fix strategies) across languages. For example, considering commonalities, we found that async wait and concurrency are very prevalent in most languages (not only Java). Considering differences, we found that in C, in contrast to other languages analyzed, async wait is *not* very common whereas improper release of resources are very common (see Table 6).

RQ3 [Cost]. *How costly is it to resolve flaky tests?* Flaky tests are common, but it is unclear how costly is it to eradicate them once they are detected. This question addresses this concern by analyzing (1) the time from opening to closing an issue and (2) the number of messages in the discussion thread of an issue. Results indicate that most issues are resolved either very early (less than 10 days) or very late (over 100 days) and some of them involve more discussion than average for a given language. Overall, results suggest that, although some issues seem to demand less effort to be addressed (e.g., platform dependency issues) and some issues seem to be less relevant (given their long inactive time), many of the issues involved a lot of discussion and quick engagement.

This paper reports on a study to understand test flakiness across programming languages. The study focused on three dimensions: concentration of root causes and their fixes (RQ1), similarity of root causes and their fixes (RQ2), and cost of resolving issues (RQ3). This study has the potential impact of *improving analysis tools and software engineering practices* under the assumption that *test flakiness is a pressing problem to the community* (see Section 6). The artifacts produced during the study are publicly available: **https://github.com/Test-Flaky/TSE22**.

2 The Anatomy of Flaky Tests

This section presents a list of root causes of test flakiness (Section 2.1) and a list of strategies used by developers to eradicate them (Section 2.2). [21] originally proposed a taxonomy for causes and fixes and [22] later extended them. This section covers the cases that we observed in our experiments. It is worth noting that (1) all issues that we analyzed could be categorized according to one of the root causes and fix strategies proposed in the studies above, i.e., we did not find new root causes and fixes; and that (2) some root causes and fix strategies listed by Eck et al. [22] have *not* been confirmed.

Table 1: Causes of test flakiness reported by Luo et al. [21].

Name	%	Description
ASYNC WAIT (AW)	36.4	When the test execution makes an asynchronous call and does not properly wait for the result of the call before using it.
CONCURRENCY (C)	15.8	When the non-determinism observed in the test output is due to different threads interacting in a non-desirable manner (but not due to asynchronous calls from the Async Wait category), e.g., due to data races, atomicity violations, or deadlocks.
TEST ORDER DEPENDENCY (TOD)	9.4	When the test outcome depends on the order in which the tests are run. For example, a test x depends on test y if x reads from a global variable changed by y. Test y passes or fails depending on whether x is executed before y.
RESOURCE LEAK (RL)	5.4	When the application does not properly manage (acquire or release) one or more of its resources, e.g., memory allocations or database connections, leading to intermittent test failures.
NETWORK (N)	4.8	When the test depends on a network resource, which is hard to control, causing flakiness.
RANDOMNESS (R)	2.5	When the test code or application code depends on random number generators without accounting for all the possible values that may be generated. For example, a test may fail only when a one-byte random number that is generated is exactly 0.
TIME (T)	2.0	When the test depends on the system time, e.g., a test may fail when the midnight changes in the UTC time zone.
FLOATING POINT OPERATIONS (FPO)	1.5	When the test performs complex floating point operations, which can produce distinct results (modulo error bounds).
UNORDERED COLLECTION (UC)	0.5	When iterating over unordered collections (e.g., sets), the code should not assume that the elements are returned in a particular order. If it does, the test outcome can become non-deterministic as different executions may have different orderings.

2.1 Root Causes of Test Flakiness

Table 1 shows a summary of the sources of flakiness (i.e., root causes) found by Luo [21] in a study involving popular Apache projects written in Java. Column "Name" shows the name used to refer to the cause of flakiness, column "%" indicates the relative frequency of that cause, and column "Description" summarizes the problem. The sources are listed from top to bottom in decreasing order of prevalence (column %). It is important to highlight that (1) according to Luo et al., the top two causes –**Async Wait** and **Concurrency**– are responsible for ∼50% of the cases of flakiness and both causes are related to the concurrent behavior of the software and that (2) the sum under column % does not add to 100 because the categories "IO" and "Hard to Classify" were left out. We did not observe flakiness due to "IO" in our experiments.

Table 2 shows the sources of flakiness reported by Eck et al. [22] but not by Luo et al. [21] for which we found occurrences in our experiments. The source **Platform Dependency (PD)** is worth discussing in more detail. In our experiments, we found a total of 14.72% PD-related issues, a high value both in absolute and relative terms (see Table 3). Whether PD should be considered a valid source of flakiness is admittedly a matter of debate. One could interpret that, according to the definition of flakiness we used [1], PD is not a valid source because test failures are not intermittent once the developer fixes the execution environment, i.e., the test either deterministically passes (on a given platform) or it deterministically fails (on a given platform). However, one could also interpret that the test fails or passes, depending on the platform it runs. Hence, it is a flaky test. For coherence with the findings of Eck et al. [22] and the bug reports issued by developers of multiple projects, we considered PD as a legitimate source of flakiness.

Table 2: Root causes reported by Eck et al. [22] but not by Luo et al. [21].

Name	Description
PLATFORM DEPENDENCY (PD)	"Non-deterministic test failures occurring only on specific platforms (e.g., a test only failing in debug builds or on 32-bit Windows 7 systems)."
TEST CASE TIMEOUT (TCT)	Test frameworks often add time budgets to test cases. This problem occurs when execution time of a test case increased over time with the addition of new functionality (e.g., substitution of mocked functions with actual functions) leading to non-deterministic test timeouts.
TOO RESTRICTIVE RANGE (TRR)	Some applications are inherently non-deterministic (e.g., games). An assertion of a test from these applications could check whether a value falls within a range of acceptable values. As consequence, a test could non-deterministically fail if the range is too restrictive.
TEST SUITE TIMEOUT (TST)	Similar to TCT. A test suite increases over time, but developers fail to observe that. As result, the test suite can non-deterministically fail because of insufficient time budget to run the test suite.

2.2 Fix Strategies of Test Flakiness

This section illustrates common fixes that developers adopted to mitigate test flakiness. Table 3 describes the fixes, indicating their prevalence in our study. The fixes are grouped by the root causes, as discussed on Section 2.1. The highlighted cells indicate the most prevalent root causes and fixes. For example, the root causes AW, C, and PD are the most prevalent causes. The table includes the fixes catalogued by [21] and [22] for which we found manifestations in our study. Column "Cause" shows the short name of the root cause, as listed on Tables 1 and 2, and shows the prevalence of the issues of that kind. Column "Fix" indicates the strategy used to eradicate flakiness.[2] Column "Description of the Fix" shortly summarizes the fix strategy. Column "Ex." shows an example issue from a GitHub project manifesting that fix strategy and, finally, column "%" shows the prevalence of that strategy in our data set. In the following, we discuss concrete examples of the five most prevalent fixes as highlighted in column "%".

PD - Correct Directories (14.2%). This example briefly describes a manifestation of flakiness on the GitHub project hpal caused by a platform dependency issue. Developers reported two platform-related problems: (1) when executing the code on Windows, the code was using an incorrect name of the command to invoke the Node.js Package Manager (NPM): "npm" as opposed to "npm.cmd", and (2) the code was not converting path separators and end-of-line symbols when executing in a different platform. Developers indicated that the test had not been previously executed on Windows. Figure 1 shows a snippet of code illustrating how these problems were addressed in code. For the first problem, a new variable npmCmd was created to store the correct name of the command, irrespective of the platform. For the second problem, a normalization function was created to translate paths referred in code. We omitted the references to

2. In case [21] and [22] used different names for a given fix, we chose to use the name that seemed to best represent that strategy.

Table 3: Common Fixes for Test Flakiness, grouped by Root Causes.

Cause	Fix	Description of the Fix	Ex.	%
AW(19.4%)	Add/modify waitFor [21], [22]	The term "waitFor" refers to an abstract operation to make the thread wait until certain desirable condition holds.	[25]	10.4
	Add/modify sleep [21]	Add or modify time delay–using a sleep command–to wait for some task to finish before proceeding to other dependent task.	[26]	6.6
	Reorder execution [21], [22]	Reorganize the statements in test case such that sufficient time is given for the results of async calls to become available before using them.	[27]	2.2
C (16.2%)	Add/modify waitFor [22]	As explained above, but the "waitFor" is added in the application code, not the test code.	[28]	3.6
	Change cond. [21], [22]	Modify control flow to enforce that certain threads follow specified execution paths.	[29]	3.8
	Protect regions [21], [22]	Identify and protect critical regions to ensure mutual exclusion of competing threads.	[30]	4.2
	Other [21], [22]	Modify code to avoid non-determinism using a strategy different than those listed above. For example, eliminate concurrency altogether, enforce certain deterministic orders between thread executions, etc.	[31]	4.6
PD (15.6%)	Add/modify tests [22]	Create new or adapt existing test cases according to the platform it is supposed to run.	[32]	1.4
	Correct directories [22]	The test case is revised to take into account the differences in the paths on each platform.	[33]	14.2
RL (10.6%)	Release resource [21], [22]	Invoke operation to release resource (e.g., memory, file, database or network connection, etc.).	[34]	10.6
N (8.2%)	Add/modify mocks [21]	Rewrite the test to use mocks (e.g., to replace a remote service).	[35]	2.4
	Add/modify waitFor [21]	In the presence of network instability, this repair adds code to retry accessing the remote resource for a fixed number of times.	[36]	5.8
TOD (7.2%)	Setup/cleanup state [21]	Flaky tests are fixed by setting up or cleaning up the state shared among the tests.	[37]	6.8
	Remove dependency [21], [22]	Flaky tests are fixed by making local copies of the shared variable to remove the dependencies on it.	[38]	0.4
TCT (5.8%)	Increase timeout [22]	"The increase of the timeout time is, obviously, the most frequent solution to this type of flakiness."	[39]	5.8
UC (5.0%)	Not specific ordering [21]	To write tests that do not assume any specific ordering on collections unless an explicit convention is enforced on the used data structure.	[40]	5.0
R (4.4%)	Control the seed [21], [22]	"The developers should control the seed of the random generator such that each individual run can be reproduced." [21]	[41]	3.4
	No Math.Random [22]	Calling Math.random is unreliable. Replace it with a random number generator that accepts an input seed.	[42]	1.0
T (3.8%)	Reduce time impr. [21], [22]	Make test more resilient to time and data representations.	[43]	3.8
FPO (2.8%)	Modify assertions [21]	Rewrite test assertions to become independent of floating-point results, which can be imprecise and lead to intermittent test failures.	[44]	2.8
TST (0.8%)	Split test suite [22]	Split the test suite that is executed in parallel in smaller chunks to reduce risk of hitting time budget.	[45]	0.6
	Skip non-init. part [22]	"Code added to skip non-initialized parts to make the test run faster and not timeout."	[46]	0.2
TRR (0.4%)	Calibrate assertion [22]	Test fails because assertion range is too rigid. Calibrate range of values used in assertions.	[47]	0.4

```
~~~~ lib/commands/new.js
-  const subproc = ChildProcess.spawn('npm', ['init'], { cwd });
+  // Node does not support PATHEXT on Windows
+  const npmCmd = process.platform === 'win32' ? 'npm.cmd' : 'npm';
+  const subproc = ChildProcess.spawn(npmCmd, ['init'], { cwd });
~~~~ test/index.js
+  const normalize = (str) => {
+    // Naively normalizes string output for OS differences:
+    // backslashes to foreslashes (paths) and the OS end-of-line to \n.
+    return str && str.replace(/\\/g, '/').replace(RegExp(Os.EOL, 'g'), '\n');
+  }; // use normalize on strings
```

Figure 1: PD, Correct Directories [33].

the normalization function for the sake of space. Note that, although both changes affect test execution, only the second change was made within the test code.

RL - Release Resource (10.6%). libvmod-dynamic is a C module of the system varnish director for the dynamic creation of backends. Figure 2 shows the fix developers created when they noticed non-deterministic test outputs. Developers found that the reason for the bug was a use-after-free problem in the function dynamic_stop(). (They used Clang's static analyzer to spot the problem.)

```
~~~~ vmod_dynamic.c
-    dynamic_free(NULL, dom);
         VTAILQ_REMOVE(&dom->obj->purged_domains, dom, list);
+        dynamic_free(NULL, dom); ...
    /* Backends will be deleted by the VCL, pass a NULL struct ctx */
+   VTAILQ_FOREACH_SAFE(dom, &obj->purged_domains, list, d2) {
+        VTAILQ_REMOVE(&obj->purged_domains, dom, list);
+        dynamic_free(NULL, dom);
    }
```

Figure 2: RL, Release Resource [48].

AW - Add/Modify waitFor (10.4%). This example describes a fix of an Async Wait flaky test in the JavaScript project pwa-studio. The test was fixed with the modification of an existing "waitFor" construct. According to Luo et al. [21], "waitFor" denotes a set of methods used to either let the current thread busy wait for some condition to become true or block the current thread until being explicitly notified. Developers mentioned the following in the discussion of the issue: *"If you run this test a few times it will be flaky without waiting an arbitrary time for a promise to finish resolving."* [25]. To circumvent the problem, developers used the NPM wait-for-expect library, whose function waitForExpect returns a Promise. The effect of using the await operator on that Promise object is to proceed with execution only when the expectations encapsulated in the promise are satisfied.

TOD - Setup/Clenup state (6.8%)

In the following, we describe an issue from a library used to run Python tests in parallel [37]. xdist [49] is an extension of the popular Python testing framework pytest [50] that enables parallel execution of test cases. It has been previously reported that unplanned parallelization of test case runs can result in flakiness [51]. TOD refers to the scenario when there is a change in the execution order of two or more tests that access the same part of the global state and that results in intermittent failures. Considering this issue, the developer of the xdist library reported "test A passes but crashes

at teardown; test B is seen as failing but hasn't been executed at all. "Then, (s)he completes with a diagnosis "When a node crashes, it pops the running item and reschedule it on a new node. However, when the node crashes *at teardown*, the running item has already been dequeued." The developer proceeded as follows to fix the issue: "I created an explicit event runtest_protocol_complete which is sent by workers after runtestprotocol (protocol refers to fixture method such as setup, call, and teardown), this way we ensure that a worker is indeed done with an item.".

AW - Add/Modify Sleep (6.6%). This example describes a manifestation of test flakiness from the Python project rapid-router caused by the slow response time of Selenium to process a request. Selenium is a very popular framework for testing web applications. The test does not take into account the fluctuation of the network load and occasionally fails. Figure 3 shows a function invoked by the test that invokes the selenium.get operation to drive the web browser to download a web page, which is then rendered on the screen. The web browser needs to render the web page on the screen before the test can send events to the screen objects. However, the code does not take this into account; the test sends events to objects before the objects are displayed on the screen. Prior work found that such "lag problem" in UI testing is common [52], [53]. The fixed version of the code addresses the problem by retrying, for a fixed number of times, to download and render the operation. A 10s sleep is added between retries to wait for the rendering task to finish.

```
~~~~ game/end_to_end_tests/base_game_test.py
  def _go_to_path(self, path):
-   selenium.get(self.live_server_url + path)
+   socket.setdefaulttimeout(5)
+   attempts = 0
+   while True:
+     try:
+       selenium.get(self.live_server_url + path)
+     except socket.timeout:
+       attempts += 1
+       if attempts > 2:
+         raise
+       time.sleep(10)
+     else:
+       break
```

Figure 3: AW, Add/modify sleep [26].

3 Research Procedure

Selection of PLs. We analyzed hundreds of issues associated with flaky tests. These issues involved a total of 741 open source GitHub projects written in five programming languages: C, Go, Java, JavaScript, and Python. The primary selection criterion was *popularity* according to GitHub's yearly survey [24]. Although Go is not among the top-10 languages reported in the GitHub's yearly survey, we included it in our study because Go programs often involve asynchronous and remote communication, which could increase (or reduce, depending on the language support) flakiness due to concurrent/async communication. These languages are also important in different domains: Systems (C), Distributed computing (Go), Android and web back-end (Java), front and back-end web development (JavaScript), and machine learning (Python). These languages also provide a diverse set of abstractions to support concurrency and networking. For example, while asynchronous programming in Java requires good understanding of threads, monitors, and of how to manipulate these programming constructs through the java.util.concurrent package, in JavaScript, asynchronous computations are abstracted away using constructs like async (to fork an execution and return an Promise object [54] to the caller) and await (to wait for a Promise to be resolved within an async function). Likewise, Go provides Goroutines and channels to facilitate concurrent and asynchronous programming.

Selection of Issues from GitHub. We followed a similar procedure to Luo et al. [21] to identify a large corpus of issues related to test flakiness from open source repositories. We wrote a Python script that uses the GitHub API to search for issues (i.e., bug reports) that contain the following terms: ("flaky" or "non-det") and "test". As mentioned above, we restricted our search to issues of projects written in C, Go, Java, JavaScript, and Python. Furthermore, to make sure that we focused on issues that developers were able to properly diagnose and fix test flakiness, we also restricted the search to (1) *closed* issues, i.e., issues that have been resolved; (2) *confirmed* issues, i.e., issues in which at least one project contributor confirmed the existence of a flaky test; and (3) *buggy* issues, i.e., issues flagged as "bug" by the developer. Since our work involves human cognizance and manual inspection, we limited the number of issues to analyze. In total, we selected 1,541 issues, nearly 300 issues per language.

Method for analyzing issues. Two of the authors worked collaboratively to classify the root cause of flakiness and corresponding fix strategy for the set of 1,541 issues. To establish a shared understanding about the causes and potential fixes, for each issue, the authors used the following procedure: (1) They studied the issue's title, body, and the stack trace, if reported; (2) They checked if any contributor indicated that the test was indeed flaky; (3) They analyzed the discussion looking for hints of root causes and potential fixes, (4) They inspected all commits and pull-requests for the issue. We inspected the commits and pull-requests associated with each issue with the goal of determining the fix strategy from code. The issue number can be tracked from the commit data. In case we did not find that information, we classified only the root cause but not the corresponding fix. We were unable to classify some of the issues (to identify root causes) or associated commits/pull-requests (to identify fixes) for one of the following reasons: (1) **Duplicated (25 issues)**: An issue was found to be duplicate. For example, the following two issues report the same problem [55], [56]. In those cases, we discarded one of them; (2) **False Positive (400 issues)**: The precision of our mining script depends on the rigour of developers in using the word "flaky". Not surprisingly, many issues were discarded as they did not actually describe flakiness manifested in test runs; (3) **Ignored Test (16 issues)**: Developers occasionally address flakiness by ignoring the test case [21], [22], [52]. For instance, in a commit to the xtext-eclipse project, the maintainer added the JUnit annotation @Ignore to ignore a flaky test [57]. (4) **Hard to classify**: For some issues we were unable to identify the **root cause** (336) or the **fix strategy** (74). Table 4 provides a breakdown of the hard-to-classify issues grouped into five categories. Most of the issues labeled as hard to classify are related to a poor issue discussion, i.e., the issue does not contain enough information to identify the root cause of flakiness or the fix strategy. For instance, the issue quic-go #65 [58] has no description and one

single general comment, while the issue scope #867 [59] requires domain knowledge to be understood. The second most prevalent case for labeling an issue as hard to classify was the absence of code in the pull request. Based on the procedures we defined for analyzing the issues, even if a contributor indicates the existence of a flaky test, if no code (for fixing the alleged flaky test) is available in the pull request, we do not proceed with the labeling of the fix strategy for such an issue. Our method for analyzing issues favors precision in the labeling over a higher number of labeled issues.

Table 4: Breakdown of the issues labeled as hard to classify

Category Name	Description	Total
Poor Discussion	Lack of information about the problem (e.g., no indication from a contributor that the test was indeed flaky, no hints regarding root causes and potential fixes)	151
No code	There is no code in the pull request.	94
Long Discussion or pull request	Discussion with many comments, code, and logs made it difficult for the authors to reach a consensus on the issue classification.	86
Tangled issues	The discussion refers to multiple problems in the same GitHub issue.	69
Limited/No access	Although the discussion started in the GitHub issue, part of the discussion and/or the solution points to information that is available outside the GitHub environment (e.g., CI platforms such as CircleCI).	10
		410

Given the varying number of issues per PL in the remaining 690 issues we decided to categorize the root cause and fix strategy of 100 issues for each language. To that end, two of the authors of this paper performed an open code procedure [60]. The authors worked independently, followed by conflict resolution meetings. When these authors were uncertain about specific cases, other researchers joined the discussion to help reach consensus. This procedure took approximately five months. In the end, we classified the root cause of **591** issues. Of these, we classified the fix strategy of **500** issues. Figure 4 shows the distribution of concerns addressed by the flaky tests from the fully inspected issues (500). The plot seems to confirm our expectations, e.g., less GUI-related tests in C, more network-related tests in Go, more GUI related tests in JavaScript, etc.

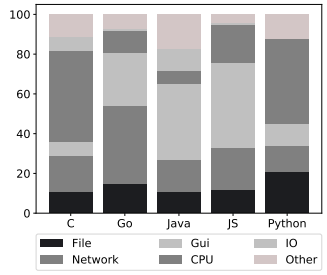


Figure 4: Distribution of concerns addressed by the test cases by domain category. The Y axis describes the amount of issue for fix, the X axis relates to PL and the bar colors the amount that each domain represents.

4 Results

This section reports the results of our study, whose central goal is to comprehend the circumstances in which flaky tests manifest across programming languages. Section 4.1 presents the raw data. The following sections elaborate on the research questions.

4.1 The raw data

Table 6 shows a summary of the root causes we observed in our data set. Each row in the table relates to a distinct root cause, as described on Section 2.1. Column "Root Cause" shows the name of the root cause. The subsequent columns show the number of occurrences of a given root cause in a PL. Column "Σ" shows the sum of cases across all languages. The rows on this table are sorted in decreasing order of "Σ". Column "#" shows the ranking of the root cause. A dash ("-") indicates that the root cause was not observed. Finally, the last two columns show, for comparison, the prevalence of that root cause as reported by Luo et al. [21] and by Eck et al. [22]. At the bottom of the table, the row "Σ" shows the totals associated with each "counter" column. In total, we were able to classify the root causes of 591 issues. The highlighted cells on a column indicate values that are one standard deviation above the mean of values on that column. For example, the mean number of occurrences of a root cause in C was 8.2 and the standard deviation of values was 8.9. For that reason, we highlighted the values above 17.8 under the column "C" on Table 6. Considering this data, note that Async Wait (AW) appears, globally, as the most prevalent root cause, in coherence with prior work [21], [52], [53]. Also note that the top 5 causes listed by Luo et al. [21] appear at the top of this ranking (as per column "#"), showing coherence of our results with theirs. Finally, note that two of the marked values were not mentioned on the referred studies. Platform Dependency was not listed as a root cause by Luo et al. [21] and Network was not listed as a root cause by Eck et al. [22]. Table 7 shows the summary of fix strategies adopted by developers when addressing flaky-related issues. The table follows a similar organization as the table for root causes. At the bottom of the table, row "Σ" shows the totals associated with each "counter" column.

A per-project analysis. Before answering our research questions we provide a brief analysis assessing the extent to which the root causes are overrepresented (or underrepresented) on the basis of the projects selected. This is important to assess the risk of a small number of projects severely influencing results of our study. The vast majority of projects contributed with only 1 issue (267 out 347). 95,7% of projects contributed with 5 or less issues (332 out 347). Similarly to the approach followed by Luo et al. [21] (Table 3, Section 3.1), we focus our per-project analysis on those projects that contributed with more than 5 issues: only 15 (4,3% of the total number of projects analyzed) — 1 for JS, 2 for Java, 3 for C, 4 for Python, and 5 for Go. Table 5 summarizes the results of our analysis. The projects analyzed are displayed in the rows, while the 13 root causes are displayed as columns. The last column "Σ" displays the total number of issues collected from a given project — for an example, a total of 8 issues were collected from the project beats, distributed across three root causes: 2 Async Wait (AW), 5 Concurrency (C), and 1 Too Restrictive Range (TRR). The row "Total" corresponds to the sum of issues for a given root cause when considering only this selection of 15 projects, while row "Other Projects" reports the sum of issues for a given root cause when considering all the projects that contributed with 5 or less issues. The row "Total Root" reports the total

Table 5: Detailed distribution of root causes for the projects with more than 5 flaky tests. Warmer colors (reddish) indicate higher values and colder colors (greenish) indicate lower values.

Repository/Project	Language	AW	C	PD	N	RL	TOD	TCT	UC	T	R	FPO	TST	TRR	Σ
quic-go	go		6		11	2		2	2		1	1			25
cdap	java	7	6		2			2	2	1			1		21
zephyr	C	2	1		3	2		4	1	1		1			15
mbedtls	C					2	1	4	1			2			10
pants	python	1	2			1		1	1	1				2	9
plaso	phyton		2			1				2		2		1	8
appium	js	2			1	1		3		1					8
beats	go	2	5											1	8
mbed-crypto	C					2	2	2		1					7
oryx	java	2		3			2								7
terraform	go	1			2				1	1				2	7
ansible	python	2			2	2									6
containerpilot	go	3	2					1							6
incubator-mxnet	python		1	2		2		1							6
weave	go				3		1	1	1						6
Total		22	25	5	24	15	6	21	9	8	1	6	1	6	149
Other Projects		**81**	**72**	**9**	**48**	**72**	**16**	**40**	**19**	**32**	**4**	**22**	**5**	**22**	**442**
Total Root		103	97	14	72	87	22	61	28	40	5	28	6	28	**591**

Table 6: Prevalence of root causes across Languages.

#	Root Cause	# Occurrences C	Go	Java	JS	Py	Σ	Ranking as per [21]	[22]
1	Async Wait (AW)	7	31	27	21	17	103	1	2
2	Concurrency (C)	15	31	25	15	11	97	2	1
3	Platform Dependency (PD)	16	8	9	24	30	87	-	7
4	Network (N)	5	31	10	16	10	72	5	-
5	Resource Leak (RL)	33	9	8	7	4	61	4	6
6	Test Order Dependency (TOD)	9	5	6	10	10	40	3	4
7	Time (T)	5	4	6	3	10	28	6	10
7	Test Case Timeout (TCT)	6	6	6	6	4	28	-	5
7	Unordered Collections (UC)	-	4	4	8	12	28	10	-
10	Randomness (R)	7	4	4	5	2	22	8	11
11	Floating Point Operations (FPO)	2	-	4	1	7	14	9	8
12	Too Restrictive Range (TRR)	1	-	5	-	-	6	-	3
13	Test Suite Timeout (TST)	1	2	-	1	1	5	-	9
	Σ	107	135	114	117	118	**591**		

Table 7: Prevalence of fix Strategies across Languages.

#	Fix Strategy	# Occurrences C	Go	Java	JS	Py	Σ	Ranking as per [21]	[22]
1	PD - Correct Directories	11	7	9	23	21	71	-	12
2	RL - Release resource	31	8	6	4	4	53	-	-
3	AW - Add/modify waitFor	6	13	15	13	5	52	1	1
4	TOD - Setup/cleanup state	7	4	6	8	9	34	3	-
5	AW - Add/modify sleep	1	14	9	3	6	33	2	-
6	TCT - Increase Timeout	6	6	7	6	4	29	-	6
6	N - Add/modify waitFor	2	9	6	8	4	29	-	-
8	UC - Not Specific Ordering	-	4	3	8	10	25	-	-
9	C - Other	3	7	8	3	2	23	5	13
10	C - Protect regions	6	7	3	2	3	21	4	7
11	T - Avoid Time Imprecision	5	1	4	2	7	19	-	15
11	C - Change condition	5	2	5	6	1	19	6	5
13	C - Add/modify waitFor	1	6	5	3	3	18	-	2
14	R - Control the Seed	6	3	2	4	2	17	-	-
15	FPO - Revise assertions	2	-	4	1	7	14	-	-
16	N - Add/modify Mocks	1	4	4	1	2	12	-	-
17	AW - Reorder execution	-	2	1	3	5	11	7	9
18	PD - Add/modify tests	4	-	-	-	3	7	-	10
19	R - No Math.Random	1	1	2	1	-	5	-	13
20	TST - Split Test Suite	-	2	-	-	1	3	-	15
21	TRR - Calibrate assertion	1	-	1	-	-	2	-	4
22	TOD - Remove Dependency	1	-	-	-	1	2	6	3
22	TST - Skip Non-Initialized Part	-	-	-	1	-	1	-	13
	Σ	100	100	100	100	100	**500**		

number of issues considered for a given root cause — for an example, when considering the root cause Async Wait (AW) we can say that out of the 103 issues analyzed, 22 were selected from this selection of 15 projects, whereas 81 issues were selected from the remaining 576 projects. Overall, we can see that even for this selection of 15 projects the collected issues were well distributed across the different root causes. Indeed, in most of the cases only one or two issues were collected for the same root cause (notice the high number of '1's' and '2's' in the table). Only two projects contributed with more than 5 issues for the same root cause: quic-go, with 6 Concurrency (C) issues and 11 Network (N) issues; and cdap, with 7 Async Wait (AW) issues and 6 Concurrency (C) issues. The 11 Network (N) issues collected from quic-go represent the maximum number of issues for the same root cause contributed by any of the projects analyzed; They account for $\approx 15\%$ (11/72) of the total number of Network (N) issues, and $\approx 10\%$ (11/107) of the total number of issues analyzed for the projects in Go. The 7 Async Wait (AW) issues collected from cdap account for only $\approx 6\%$ (7/103) of the total number of issues from that root cause. We conclude that it is not the case that the root causes are overrepresented (or underrepresented) on the basis of the projects selected.

4.2 Answering RQ1: Concentration

The first research question we posed is as follows: *Is it the case that, for a given PL, a small fraction of sources and corresponding fix strategies are associated with the majority of the issues?* The answer to this question enables researchers and practitioners to focus on a small set of causes and fixes.

Tables 8 and 9 summarize our observations about the concentration of root causes and their corresponding repairs. More precisely, these tables show the percentage of the issues that are covered at different cutoff points of the rankings of root causes and fix strategies. Let us consider Table 9, which reports on the concentration of fix strategies. Column "PL" shows the programming language, the columns under "%" show the percentage of issues covered by the top-N (N=1, 3, 5, and 10) fix strategies of a given PL. For example, considering Python, we found that 40% of the issues (Table 7, column "Python") use the three most common fix strategies adopted by developers.

Overall, results indicate that 78.07% of the issues associated with any given PL are related to five root causes. This number corresponds to the average of the values from column "top 5" on Table 8. For Go projects, we observed that 68.89% of the issues are related to the top 3 root

Table 8: Root Causes.

PL	% of issues top 1	top 3	top 5	top 10
C	30.84	59.81	81.31	98.13
Go	68.89	68.89	81.48	98.52
Java	23.68	54.39	69.30	100.00
JS	20.51	52.14	73.50	98.29
Python	25.42	50.00	84.75	97.46

Table 9: Fix strategies.

PL	% of issues top 1	top 3	top 5	top 10
C	31.00	49.00	73.00	87.00
Go	14.00	36.00	65.00	89.00
Java	15.00	33.00	48.00	76.00
JS	23.00	60.00	60.00	92.00
Python	21.00	40.00	54.00	82.00

causes. That mark is well above the mark of different PLs at the same cutoff point. The three most prevalent root causes that we observed in Go are Async Wait, Concurrency, and Network. Curiously, one important use case of the Go language is distributed and asynchronous computing [61], [62], which is related to these root causes. We conjecture that the higher number of issues involving those three root causes is a confirmation that programs in that language highly use asynchronous communication, concurrency, and the network. Results also show that 85.20% of the flaky tests can be fixed with one of the ten most common fix strategies (avg. of column "top 10" on Table 9). Note that developers used only 43.48% of the fix strategies to fix most flaky tests.

Summary RQ1: Our results show that, for any given language, most issues can be explained by a small fraction of root causes (5/13 root causes cover 78.07% of the issues) and can be fixed by a relatively small fraction of fix strategies (10/23 fix strategies cover 85.20% of the issues).

4.3 Answering RQ2: Similarity

The second research question we posed is: *How (dis)similar are the root causes and fix strategies across PLs?* Answering this question enables one to decide whether tuning analysis (e.g., flakiness detection techniques) to specific languages is a relevant problem. If, for example, results show that the causes and fixes have similar manifestations across languages, then adaptation of tools to particular PLs would be unjustified.

4.3.1 Root Causes

As Table 6 indicates, Async Wait (17.43%), Concurrency (16.41%), and Platform Dependency (14.72%) are the most prevalent root causes, overall. In Go and Java, particularly, Async Wait corresponds to 22.96% and 23.68% of the total number of root causes inspected, respectively. Table 10 summarizes the (dis)similarity of root causes across languages. We included on this table any root cause that appeared among the top 5 most prevalent on any given programming language (see Table 6). A bullet (•) indicates that the root cause is among the top 5 root causes to explain flakiness in the corresponding language. In the event of a tie in the 5th position, we added a bullet to each of the tied root causes (this is the reason why C and Python have more than 5 bullets). For example, overall, Network (N) is among the top 5 most prevalent root causes, but it is *not* frequent in C. Unordered Collections (UC), in contrast, is very common in Python, but less common in the other four languages. We sorted the rows in this table by the number of bullets. The rows at the top indicate the cases that are more similar; highlighted rows show root causes that are *prevalent in all languages*. The rows at the bottom of the table show the cases more dissimilar. It is worth noting the prevalence of Platform Dependency (PD) issues in all languages, especially considering that Luo et al. [21] did not list that root cause in their study and that Eck et al. [22] found that root cause was not highly common in their study with flaky tests from Java programs; they found PD to be the 7th most prevalent root cause. Considering only Java programs, PD was the 4th most prevalent root cause in our data set, behind AW, C, and N.

Table 10: Similarity of root causes.

	C	Go	Java	JS	Py
AW	•	•	•	•	•
C	•	•	•	•	•
PD	•	•	•	•	•
N		•	•	•	•
RL	•	•	•		
TOD	•			•	•
R	•				
T					•
UC					•

4.3.2 Fix Strategies

Table 11 reports on the (dis)similarity of fix strategies across languages. This table includes any fix strategy that appears among the top 10 most prevalent strategies on any given programming language (see language columns on Table 7). A bullet (•) indicates that the fix strategy is among the top 10 strategies to address flakiness in a given language. As in Table 10, we sorted the rows in this table by the number of bullets. The rows at the top indicate the most similar cases; highlighted rows show the fix strategies that are prevalent in all languages. The rows at the bottom of the table show the fix strategies that are most dissimilar amongst those that are prevalent on at least one programming language. For instance, Test Order Dependency (TOD) was found to be a frequent kind of root cause in C, JS, and Python, but it was *not* very common in Java. This is surprising given that the object of study of recent work on flakiness was TOD in Java projects [63].

The view on the left-hand side of the table highlights the (dis)similarity of fix strategy across languages, but it fails to quantify the discrepancy. To fill that gap, Table 11 also reports, at the right-hand side, the absolute difference —referred to as Δ— between the average ranking of a fix strategy, as listed under the column "#" from Table 7, and the PL-specific ranking of a fix strategy. For example, consider the cell associated with Go and "PD - Correct Directories". The value 4 is the absolute difference between 1, which corresponds to the rank of that fix strategy in the

Table 11: Similarity of fix strategies.

	Prevalence C	Go	Java	JS	Py	Δ C	Go	Java	JS	Py
AW - Add/modify waitFor	•	•	•	•	•	1	1	2	1	4
PD - Correct Directories	•	•	•	•	•	1	4	1	0	0
RL - Release resource	•	•	•	•	•	1	2	4	6 ⇓	7 ⇓
TCT - Increase Timeout	•	•	•	•	•	1	2	1	0	3
TOD - Setup/cleanup state	•	•	•	•	•	1	6 ⇓	2	1	1
AW - Add/modify sleep		•	•	•	•	9 ⇓	4	3	5	1
N - Add/modify waitFor		•	•	•	•	6 ⇓	3	0	3	1
C - Add/modify waitFor		•	•	•		1	5	4	3	1
C - Change condition	•		•	•		3	3	3	5	7 ⇑
C - Other		•	•	•		2	4	5	1	6
UC - Not Specific Ordering		•		•	•	12 ⇓	2	3	5	6 ⇑
AW - Reorder execution				•	•	3	3	1	7 ⇑	10 ⇑
C - Protect regions	•	•				6 ⇑	5	4	4	2
R - Control the Seed	•			•		10 ⇑	1	2	6 ⇑	1
T - Avoid Time Imprecision	•				•	3	6 ⇓	0	3	7 ⇑
FPO - Revise assertions					•	3	4	4	1	11 ⇑
N - Add/modify Mocks		•				2	6 ⇑	5	0	1
PD - Add/modify tests	•					8 ⇑	1	2	2	6 ⇑

aggregate ranking (Table 7, column "#"), and 5, which corresponds to the position of that same strategy in the ranking associated with Go (obtained from Table 7, column "Go"). We highlighted the cells whose values of Δ are above 5; these are the cases with higher discrepancies. The direction of the double arrow in those cells indicates the increase ($\Uparrow$) or decrease ($\Downarrow$) in the frequency of observations of that fix strategy relative to the aggregate ranking. The number of up and down arrows provides a rough indication of how much a language differs from the average. For example, compared to the average ranking (see Table 7), Java is the least discrepant whereas Python is the most discrepant. Section 5 discusses notable cases. For example, we observed that "UC - Not Specific Ordering" was rarely used in Java and C, whereas "AW - Add/modify sleep" was rarely used in C.

Summary RQ2: Results indicate that a small set of root causes and fix strategies are common regardless of the programming languages, but some languages have their specificity. For instance, the root cause RL is particularly prevalent in C whereas the root cause N is particularly prevalent in Go (and T and UC in Python).

4.4 Answering RQ3: Cost

The third research question we posed is as follows: *How costly is it to resolve flaky tests?* This question focuses on cost, which is a more qualitative aspect compared to prevalence and similarity. We started our analysis inspecting two metrics: (M1) number of days between opening and closing an issue and (M2) number of comments in the discussion thread within that time interval. It is worth noting that assessing resolution cost and importance of an issue is challenging as it depends on a number of factors. For example, too long resolution time may indicate that a task has been neglected for its lower importance as opposed to indicate the contrary, i.e., that fixing the flaky test is intrinsically complex and time-consuming. Nevertheless, these metrics help us acquiring a grasp on resolution cost and importance. For comparison with the issues related to flaky tests, we mined 1K "non flaky" issues for each programming language. The extraction of those metrics is automated through the GitHub API. Table 12 shows results for M1 at the top and results for M2 at the bottom. Rows "Flaky" and "Other" list, respectively, results associated with flaky test issues and other issues. The key observations from these results are that: (1) Compared to "Other", issues related to flaky tests take longer to resolve and involve more discussion, (2) C, JavaScript, and Python showed the highest difference in resolution time between flaky and other issues, and (3) C and JavaScript showed the highest difference in number of comments.

Table 12: Number of days and comments until issue is closed.

	C	Go	Java	JS	Py
M1 - Number of days					
Flaky	123	58	61	77	86
Other	14	14	19	15	20
M2 - Number of comments					
Flaky	10	4	5	11	6
Other	3	1	1	1	2

Figure 5 shows the histograms of resolution time on each analyzed language for the flaky and "non flaky" test issues. The size of the *dark-colored bar* indicates the percentage of issues closed within a given time interval whose discussion threads contained a number of comments below the average for the respective language. Likewise, the size of the *light-colored dashed bar* indicates the percentage of closed issues with number of comments above the average. For the flaky test issues, the key observations are that, most notably in C, many issues seem to be resolved late after it is opened, suggesting that those problems are not all that relevant (observe the red line at the time interval ">100"). However, surprisingly, many of those issues have an above-average number of comments, indicating that they were not trivially resolved. More importantly, results indicate that, for any given language, many issues are resolved a few days after they are opened, suggesting they are important, and some of them involve more discussion than average, especially in C, Java, and Python. We also noticed that flaky tests in JavaScript involved less discussion, overall, suggesting they were easier to address. In fact, most JavaScript issues are related to Platform Dependency, which we indeed found to typically require simple fixes (e.g., revise paths). Regarding the "non flaky" test issues we observed that they are generally resolved faster than the flaky test issues in the same project and, for all the languages analyzed, that most of the issues are resolved a few days after they are opened (notice the sharp drop of the red line in all the plots after the time interval "10").

Table 13 shows, for different pairs of fix strategy and language, the number of issues resolved within 60 days whose discussion threads contained more comments than average. The rationale for the selection was to focus on the cases that are presumably more complex (as per number of discussions) and more pressing (as per time to address) to resolve. Overall, we observed that the fix strategies did not change much, compared to Table 7, suggesting that analyzing issues based on prevalence and on cost resulted in similar observations.

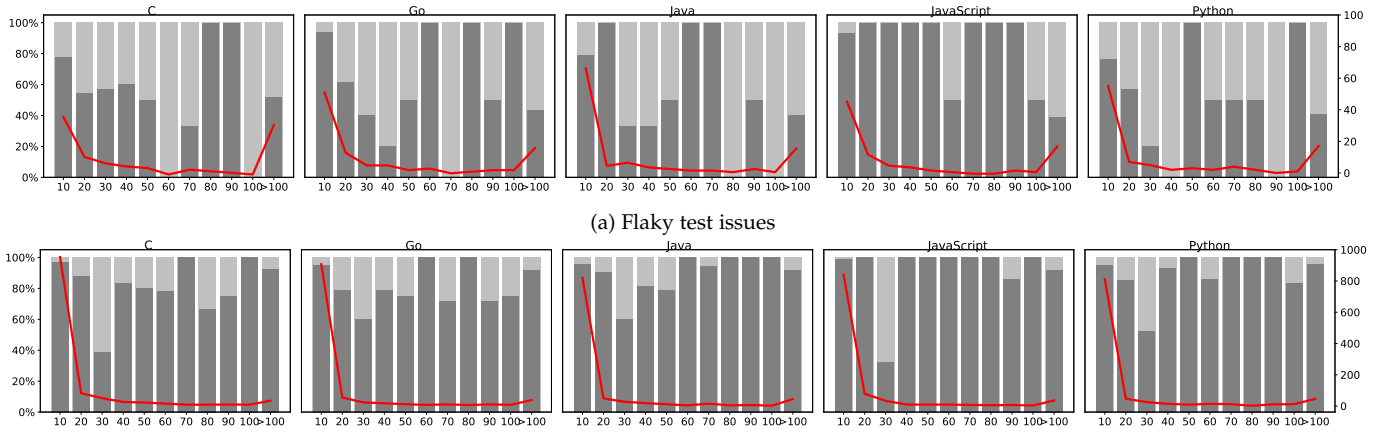
Table 13: Number of issues with resolution time no longer than 60 days and number of comments above average.

Fix Strategy	C	Go	Java	JS	Py	Σ
PD - Correct Directories	1	2	3	2	5	13
AW - Add/modify waitFor	2	3	2	-	2	9
RL - Release resource	6	-	-	-	2	8
AW - Add/modify sleep	-	2	2	1	2	7
C - Other	1	2	1	-	1	5
N - Add/modify waitFor	-	3	1	-	1	5
TOD - Setup/cleanup state	2	-	1	-	2	5
TCT - Increase Timeout	-	-	2	-	2	4
C - Protect regions	2	1	-	-	-	3
PD - Add/modify tests	2	-	-	-	-	2
T - Avoid Time Imprecision	1	-	1	-	-	2
N - Add/modify Mocks	-	1	1	-	-	2
AW - Reorder execution	-	-	-	-	2	2
FPO - Revise assertions	1	-	-	-	-	1
R - Control the Seed	-	1	-	-	-	1
C - Change condition	-	-	1	-	-	1
UC - Not Specific Ordering	-	-	-	1	-	1
C - Add/modify waitFor	-	-	1	-	-	1

Given that the time from opening to closing an issue can be highly influenced by the characteristics of individual projects, we also performed an additional study on the costs to resolve flaky tests on 5 projects; one project per programming language. For each PL, we selected the project with the highest number of flaky tests: quic-go, cdap, zephyr, pants, and appium. Details for these projects are presented on Table 5. Table 14

Table 14: Number of days and comments until issue is closed (per-project analysis).

	C **zephyr**	*Go* **quic-go**	*Java* **cdap**	*JS* **appium**	*Py* **pants**
M1 - Number of days					
Flaky	213	32	7	223	142
Other	24	3	1	1	1
M2 - Number of comments					
Flaky	17	2	3	49	8
Other	1	3	1	2	2



(a) Flaky test issues

(b) "non flaky" test issues

Figure 5: Histograms of distributions of resolution time for flaky and "non flaky" issues. The x-axis shows the time interval in number of days. For a given language and time interval, the *dark-colored bar* and the *light-colored dashed bar* indicate the percentage of issues closed whose discussion threads contained a number of comments below and above the average, respectively (left y-axis). The red line (right y-axis) represents the total number of issues closed in each time interval for a given language.

shows the costs to resolve issues on the projects analyzed based on the number of days (M1) and number of commits (M2) until the issue is closed. Overall, we can see that the results are similar to those observed when considering the data grouped by PL, i.e., when compared to "Other", issues related to flaky tests take longer to resolve and involve more discussion. The highest difference in resolution time between flaky and other issues is observed for the projects appium and pants.

Summary RQ3: Overall, the data shows that many flaky-related issues are resolved early (suggesting importance) and many of those issues involve more discussion than average (suggesting complexity). Results provide initial evidence suggesting that resolving flakiness is costly sometimes.

5 Discussion

This section discusses the data presented in the previous sections.

Comparison with Luo et al. [21] and Eck et al. [22] Luo et al. and Eck et al. studied test flakiness on projects written in Java. We found, perhaps not unexpectedly, that the ranking of root causes we obtained for Java was similar to the one reported by Luo et al. [21]. For example, the top 5 root causes reported by Luo et al. [21] overlap with the top 6 root causes that we found for Java. The difference between these two sets was the root cause "Platform Dependency", which Luo et al. [21] did not consider. In contrast, when comparing our ranking with that of Eck et al. [22], we observed important differences. For example, we found very few cases of "Too Restrictive Range", which, strangely, appeared as the 3rd most prevalent root cause in their data set. To recall, that fix is used when the developer observes that an assertion should be more permissive as to admit a wider range of possible values (see Table 2). Curiously, most cases of this root cause we found appeared in issues of Java projects; 5 of the 6 cases. The most notable discrepancy that we observed between our findings compared to prior work was the prevalence of flakiness due to "Platform Dependent" issues. That cause appeared at the top of the ranking in JavaScript and Python and was relatively common in C (2nd) and Java (2nd). That root cause has only one associated fix strategy, which appeared as the most prevalent strategy when considering the aggregate ranking (see Table 7). Luo et al. [21] did not mention this category in their paper, perhaps because they considered the case to be manifestations of non-portable code as opposed to a legitimate source of flakiness. Note that the execution environment needs to change in order to observe the instability. Eck et al. [22] identified this source of flakiness. In contrast to our findings, however, they found that it was not very common. This is surprising given that 14.2% of the 591issues we analyzed are of this kind and they appear in all PLs we analyzed. Despite their high frequency, we noted from the discussion threads that the flakiness of this kind are easy to diagnose and fix by the developer.

Distribute computing with care (Go). We observed that flaky tests due to Async Wait, Concurrency, and Network issues are highly common in all languages. However, compared to other languages, Go shows a substantially higher concentration of these three root causes. A total of 68.89% of the flaky tests analyzed in Go are related to these root causes. Perhaps the reason for this concentration is related to the fact that the most common use cases of Go are in distributed computing. For instance, Go provides channels and Goroutines to facilitate concurrent and asynchronous programming. Ray et al. [64] conducted a popular study relating programming languages and code quality, including bug proneness. It is worth noting that this study does *not* attempt to compare robustness of programming languages to test flakiness. Instead, our focus is on understanding the test flakiness phenomena across programming languages. (Section 6 discusses implications.)

Do no block the Web (JS). Async Wait issues are the most predominant root cause of flaky tests, overall. In that category, "Add/Modify waitFor" and "Add/Modify sleep" were the fix strategies employed more frequently. Interestingly, we found that JavaScript developers employed

almost exclusively the "Add/Modify waitFor" idiom, which is typically implemented using asynchronous constructs. We believe there are two reasons for that: 1) Until ECMAScript 8 (released in 2017), JavaScript did not have a native sleep function that would enable the adoption of "Add/Modify sleep"; By means of comparison, Promises were introduced in 2015 and async/await constructs were introduced in 2016; 2) async/await constructs are broadly encouraged. It was not uncommon to find issues showing the intent to refactor *all* test cases to use async/await (e.g., [65]).

Suboptimal approaches to fix flakies. Although "waitFor" can be implemented in various ways, we found that "busy wait(ing)" is the preferred choice of developers as it is easy to implement [66]. However, "busy wait" unnecessarily burns CPU cycles and its adoption can become problematic when (1) it is important to speed up regression testing [51] and (2) it is broadly used in a test suite to circumvent flakiness. Still related to Async Wait, we noticed a large number of fixes that add or increase "sleep" time on test cases as to avoid the test code accessing a resource before that resource is ready to be used. Although that strategy does not burn CPU cycles (as the running thread is suspended) the effect is similar to busy waiting, i.e., it slows down regression testing. We conjecture that developers are anxious to address flakiness and choose the solutions that can be implemented the fastest. It is worth noting that libraries to handle async waits exist. For example, the Awaitility Java library [67] is designed to synchronize asynchonous operations and is frequently used to coordinate operations in UI testing, which is well-know to manifest flakiness related to problems such as accessing UI widgets before they are properly rendered on screen [52], [53]. The fix strategy "C - Other" is also rather common, corresponding to 28.40% of the fixes related to Concurrency. To recall, this fix strategy uses a mechanism other than those listed on Table 3 to handle flakiness due to concurrency. For example, in the project bajel [68] developers reported a race condition that occurs when a file is not available during a directory scan. Developers observed that an exception is raised whenever the race is manifested, but instead of fixing the code to prevent the race developers ignore the test execution if the race occurs. To circumvent the problem, developers wrapped the code raising the exception in a try block associated with an empty catch block. If a race condition occurs, an exception is raised, then captured by the catch block, and the test is considered to pass.

Long duration issues (C). On average, a flaky issue is fixed in 80.9 days. However, flaky issues in C are fixed in 123 days on average (1,5× longer than the average). We found 31 issues in C that took more than 100 days to be fixed. When analyzing those issues, we noticed dormant behavior [69], i.e., the issues remained opened for a long time –often years– waiting someone to fix it. An example of this is the issue #1099 from the stellar-core project, which did not receive one single comment during a period of two years. Although some issues in C were actively discussed and received many fix attempts, they still required years to be properly fixed and closed. One example is issue #1129 from the mbedtls project, which reported a Release Resource bug, and received 40 comments, but took nearly a year to be closed.

6 Implications

elaborate We detail the implications of our findings in the following.

Practice. The answers to our research questions are important for testers writing code in a certain language. For example, when observing that a given test manifests non-deterministic behavior, developers should focus their attention to certain root causes that are more prevalent for the language under use. Once a root cause is identified, the scope of possible fix strategies can also be reduced. For JavaScript code, we found that when an Async-Wait (AW) problem is detected, the probability that an "Add/modify waitFor" solution is used to repair the problem is much higher compared to the probability of using the two other AW solutions. Of course such implications hold only if the reported results can be generalized, which can only be achieved by conducting additional studies.

Research. Our findings can also enable the development of better techniques for the analysis of flakiness. For example, researchers could leverage our findings to propose specialized *test repair tools*. A test repair tool for JavaScript would recommend developers to adjust the timeout of a test case ("TCT - Increase Timeout") with high probability and would recommend with lower probability the increase of sleep time ("AW - Add/modify sleep"). Naturally, additional test data, such as execution logs, should be used to drive the repair. Likewise, *static flaky test detectors* [9]–[12], [23] could discriminate root causes that are more relevant to a certain language with the goal of improving precision at the expense of a marginal loss in recall. These tools use text classifiers created from canonical representations of test cases. For example, we observed that 68.89% of the issues in Go are related to only three root causes and 65% of the flaky tests were fixed with five (out of 23) different fix strategies. One route to improve precision is to define custom features to identify the most common root causes from a given language. Considering Go, for instance, the classification model of a flaky detector could be augmented with three extra features that identify the presence of the three most prevalent root causes of flakiness. For that, one can write a set of patterns to identify asynchronous computation, concurrency, and network accesses in the test cases of Go projects. To sum up, the information we collected can provide helpful guidance for tools that make probabilistic decisions. Test repair tools and flaky detectors are two good examples of such tools. We considered the development of such tools to be out of scope of this work.

7 Threats to Validity

This study has different threats to validity. First, we could have made mistakes in the categorization of the root causes and fix strategies. Two authors of this paper led the effort of analyzing the many issues. The process was time-consuming—spanning several months—and was not immune to errors as it is based on human interpretation. To mitigate that threat, meetings with all authors were held twice a week to discuss selected issues, each of the two "inspector" authors analyzed each issue separately, and discussed the cases where there was initial disagreement to

reach consensus. Second, the quality of the text describing issues could have influenced the judgement of our inspectors. To mitigate that threat, we discarded issues for which we were not confident. It is worth noting that using issues as the object of our study was key to 1) analyze flaky tests with higher confidence and 2) to scale our study to a large number of cases. Considering 1), maintainers possess technical and domain knowledge that we cannot match. They are more likely to understand the outcome of a test run. Also, the studied projects oftentimes rely on continuous integration environments, which help maintainers to easily pinpoint tests executions that are likely to be flaky. Considering 2), it would be impractical and ineffective to analyze ourselves the root causes and fixes from each project. Our data set includes a total of 591projects. To independently assess root causes and implement fixes, we would need to configure the project, execute the test suites, inspect the test outcomes, diagnose flakiness, and implement the fixes. The maintainers of these projects have already performed all those tasks. We focused on digesting the information already available in the discussion threads of issues and their corresponding commits. Third, our findings are restricted to the data set of issues we mined. Section 3 describes the rationale we used to define our search criteria. For example, the choice of search query was made with the goal of reducing false positives. Also, we used a cap on the number of issues for the sake of time. After five months of analysis, we ended up discarding 400 false positives. (25.95% of the total). Nevertheless, in terms of size of the dataset, our dataset greatly expands the dataset of related works that leverage qualitative analysis to find flakiness causes and solutions. For example, Luo et al. [21] analyzed 201 commits in 51 Java projects, Eck et al. [22] curated 200 flaky tests previously fixed and asked developers to classify them, Thorve et al. [70] studied 77 commits in 29 Android projects, and Romano et al. [53] found 235 flaky UI tests in 62 projects. Finally, we focused on a small selection of programming languages. Our findings are restricted to that set. Although different languages can bring new surprises, we believe that the central message of our study remains even with the inclusion of other languages. For example, we observed commonalities and differences in root causes and fix strategies across languages. Fourth, our findings for RQ3 may have been influenced by project-specific characteristics influencing the time from opening to closing an issue. To mitigate this threat, we performed an additional study on the costs to resolve flaky tests on 5 projects – one per language– and the results were similar to those observed when considering the data grouped by PL, i.e., when compared to "Other", issues related to flaky tests take longer to resolve and involve more discussion.

8 Related Work

Empirical studies. Several studies were conducted to characterize flaky tests. To the best of our knowledge, Luo et al. [21] were the ones that first attempted to classify the source of flakiness and the strategies to fix them. Their work is based on 201 commits made to ∼51 Apache projects. Among their contributions, they presented a taxonomy of 10 sources of test flakiness. The authors also observed that the most common reason is related to Aync Wait and Concurrency (see Table 1). Thorve et al. [70] conducted a study about test flakiness focused on 29 Android apps and observed that the causes of flakiness in Android apps are similar to those found by Luo et al. [21]. More recently, Ray et al. [53] also studied flaky tests in UI tests, a category of tests that others [52], [71] also found to manifest lots of flakiness. They analyzed 235 flaky test samples from Android apps (83) and web projects (152). By also using the taxonomy provided by Luo et al. [21], the authors found that the majority of flaky tests are due to Async Wait code, a finding also observed in this study. Although their work also explored multiple programming languages (e.g., JavaScript, TypeScript, Kotlin, and Java), the focus of the paper was the UI; the paper did not attempt to analyze similarities and differences of flakiness manifestations across programming languages. The work of Vahabzadeh et al. [72] focused on bugs on test case. When investigating 499 buggy test cases, the authors observed that ∼21% of them are actually flaky ones. Among these flaky tests, the most common ones were related to Async Wait, Race Condition, and Concurrency Bugs. Eck et al. [22] curated a corpus of 200 flaky tests, and then asked the developers that fixed these tests to provide their rationale regarding their fix strategies. In this study, the authors also reported that Async Wait and Concurrency are the most common reasons for flakiness. Similar to Eck et al. [22], Ahmad et al. [73] conducted a survey with practitioners and accessed the codebase of two software companies aimed to catalog root causes and fix strategies of test flakiness. Async wait was also the most common root cause for flakiness. Lam et al. [74] studied the evolution of 55 Java projects to determine when a flaky test is introduced (or observed) in the codebase. Among the findings, the authors observed that the majority of the flakies (75%) are introduced at the moment they are created; this proportion increases to 85% when considering new tests and existing (yet modified) tests. Most of these studies though focus on Java programming language and its ecosystems. More recently, Gruber et al. [75] studied the presence of flaky tests in Python. The authors ran the tests of 22k Python projects for 400 times and observed that flaky tests in Python are as commonplace as in Java, happening in 0.8% of the studied test cases. Given the method used in the work to identify flakiness led to the inflation of flakies detected due to Test Order Dependency (TOD), it is not valid to compare the results they obtained with ours. Two hundred reruns of the 400 of each test suite execute test cases in different orderings, an approach similar to that adopted by iDFlakies [76]. Not surprisingly, the authors observed that the majority of the tests are flaky due to TOD.

Detection Techniques. The problem of *automated detection of test flakiness* was intensively investigated in research [9]–[13], [15], [16], [23], [74], [76], [77]. Gambi et al. [77] proposed a practical approach, based on flow analysis and iterative testing, to detect flakiness due to broken test dependencies. Shi et al. [15] proposed iFixFlakies to find and fix flaky tests caused by broken test dependencies. Bell et al. proposed DeFlaker [13], a dynamic technique that monitors the latest code changes and marks any new failing test that did *not* execute changed code as flaky tests. Dong et al. [16]

proposed FlakeShovel, a tool to detect flakiness in Android apps by monitoring and manipulating thread executions to change event orderings. Machine learning approaches have also been used to try to predict flakiness. King et al. [10] used Bayesian networks for flakiness classification. Pinto et al. [11] used classical machine learning algorithms and NLP techniques to classify flaky test cases. Alshammari et al. [23] improved their classication models by using additional features. Haben et al. [78] replicated the work of Pinto et al. [11] on Python projects and by also extending the set of features they used. Overall, these studies provide evidence that static detection of flakiness using ML classification is promising. However, further work needs to be done to avoid overfitting the models to the training data sets. Although our work does not directly contribute with new techniques to detect flakiness, our findings (e.g., focusing on a subset of fix strategies covers a good proportion of the causes) could help researchers design better detection tools. Lam et al. [74] recently conducted a longitudinal study to observe when –during evolution– flakiness could be observed in tests. They found that 75% of the cases of flakiness could be detected when the test is created and another 10% of the cases could be detected when the test is modified. Based on those findings, a development team could configure continuous integration pipelines [79] to provision the execution of flakiness detectors on tests created or modified; thus, amortizing the detection cost and reducing the chances that flakiness is detected at failure time, when the pressure to deliver fixes is much higher.

9 Conclusions

Flaky tests are a serious problem to the effectiveness of regression testing. Several empirical studies have been conducted to comprehend the flakiness phenomena, but they focus predominantly in Java. The goal of this paper is to understand the relationship between the programming language and the flakiness phenomena. We selected languages to analyze that are highly popular and that deal distinctly with concurrency and asynchronicity, which have been shown to be linked to flakiness. We observed e.g. that 1) a small fraction of root causes can explain most cases of flakiness, 2) a small fraction of fixes are used to circumvent flakiness, and 3) there is a high similarity of root causes and fixes across languages. It is worth noting that the generality of these findings is limited by the set of projects and languages that we analyzed. The artifacts produced during this study are publicly available: **https://github.com/Test-Flaky/TSE22**.

Acknowledgment

This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0. Keila was supported by a FACEPE fellowship number IBPG-1316-1.03/19.

References

[1] J. Micco, "Flaky tests at google and how we mitigate them," 2016, https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

[2] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*, 2019, p. 101–111.

[3] J. Listfield, "Where do our flaky tests come from?" 2017, https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[4] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *ICSE*, 2015, p. 483–493.

[5] "Github: Reducing flaky builds by 18x." 2020, https://github.blog/2020-12-16-reducing-flaky-builds-by-18x/.

[6] M. Harman and P. W. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *SCAM*, 2018.

[7] C. Developers, "Chromium flakiness dashboard howto," http://www.chromium.org/developers/testing/flakiness-dashboard, 2021.

[8] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020. [Online]. Available: https://books.google.com.br/books?id=TyIrywEACAAJ

[9] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE*, 2015, pp. 39–48.

[10] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *QRS-C*, 2018, pp. 100–107.

[11] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR*, 2020, pp. 492–502.

[12] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[13] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: automatically detecting flaky tests," in *ICSE*, 2018, pp. 433–444.

[14] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019, pp. 312–322.

[15] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE*, 2019, pp. 545–555.

[16] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "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*, 2021, pp. 367–378.

[17] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests," 2019, https://labs.spotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

[18] J. Micco, "The state of continuous integration testing @google," 2017, ICST.

[19] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *ICSME*. IEEE, 2018, pp. 534–538.

[20] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *ESEC/FSE*, 2018, p. 857–862.

[21] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014, p. 643–653.

[22] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *ESEC/FSE*, 2019, p. 830–840.

[23] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," in *ICSE*, 2021, pp. 187–187.

[24] "The state of the octoverse," 2020, https://octoverse.github.com/.

[25] "fix: Correct failing test with wait-for-expect," 2021. [Online]. Available: https://github.com/magento/pwa-studio/pull/738

[26] "Selenium tests hang," 2021. [Online]. Available: https://github.com/ocadotechnology/rapid-router/issues/830

[27] "Set the signal mask before forking," 2021. [Online]. Available: https://github.com/box/ClusterRunner/issues/180

[28] "Test consensus leader gets votes before next block is flaky," 2021. [Online]. Available: https://github.com/orbs-network/orbs-network-go/issues/197

[29] "Flaky test failures," 2021. [Online]. Available: https://github.com/elastic/beats/issues/1517

[30] "Data races," 2021. [Online]. Available: https://github.com/hashicorp/packer/issues/42

[31] "Flaky test in atom/atom," 2021. [Online]. Available: https://github.com/atom/atom/issues/6274
[32] "Flaky tests on macos," 2021. [Online]. Available: https://github.com/aaugustin/websockets/issues/241
[33] "Support windows," 2021. [Online]. Available: https://github.com/hapipal/hpal/pull/36
[34] "fix: flaky test archivertestcase," 2021. [Online]. Available: https://github.com/borgbackup/borg/issues/5196
[35] "use a chan to store sent packets in mock connection," 2021. [Online]. Available: https://github.com/lucas-clemente/quic-go/pull/795
[36] "iphone test is flaky," 2021. [Online]. Available: https://github.com/hashicorp/terraform/pull/12397
[37] "Identify correct test crashed during teardown and support multiple test logs from plugins," 2021. [Online]. Available: https://github.com/pytest-dev/pytest-xdist/pull/218
[38] "Test failures in ios test app on actual devices," 2021. [Online]. Available: https://github.com/thaliproject/jxcore/issues/105
[39] "[tests](fix): Fix flaky test timeout," 2021. [Online]. Available: https://github.com/scality/Arsenal/pull/111
[40] "Fix flaky recreateindex test," 2021. [Online]. Available: https://github.com/mitodl/open-discussions/issues/1724
[41] "Fix integration and junit tests," 2021. [Online]. Available: https://github.com/malawski/cloudworkflowsimulator/issues/25
[42] "Fix slow metrics provider tests," 2021. [Online]. Available: https://github.com/FormidableLabs/nodejs-dashboard/issues/80
[43] "flaky test symantec," 2021. [Online]. Available: https://github.com/log2timeline/plaso/issues/54
[44] "Flaky test: Rdfspeedit.testrdfspeedregression," 2021. [Online]. Available: https://github.com/OryxProject/oryx/issues/194
[45] "Make test suite faster and less flaky," 2021. [Online]. Available: https://github.com/pyro-ppl/pyro/issues/114
[46] "Fix flaky checkbox end-to-end test," 2021. [Online]. Available: https://github.com/uber/baseweb/issues/229
[47] "Fix metrics tool on core-lib/benchmarks/all.ns," 2021. [Online]. Available: https://github.com/mockito/mockito/pull/87
[48] "Fix 9 in two parts," 2016. [Online]. Available: https://github.com/nigoroll/libvmod-dynamic/pull/11
[49] H. K. (and contributors), "pytest-xdist," https://pypi.org/project/pytest-xdist/, 2021.
[50] "pytest: helps you write better programs," https://docs.pytest.org/en/stable/, 2021.
[51] J. Candido, L. Melo, and M. d'Amorim, ""test suite parallelization in open-source projects: A study on its usage and impact"," in *ASE*, 2017, pp. 838–848.
[52] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *ICSME*, 2020, pp. 301–311.
[53] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *ICSE*, 2021, p. 1585–1597.
[54] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," *SIGPLAN Not.*, vol. 23, no. 7, p. 260–267, 1988.
[55] "next.clienttest failing," https://github.com/TheScienceMuseum/collectionsonline/issues/1022, 2021.
[56] "multiple-filters.clienttest failing," https://github.com/TheScienceMuseum/collectionsonline/issues/1021, 2021.
[57] "Ignored test," https://github.com/eclipse/xtext-eclipse/commit/567ba325545cea2ab85319df82e0249f42523613, 2021.
[58] "Server tests flaky in travis," https://github.com/lucas-clemente/quic-go/issues/65, 2021.
[59] "Deal with starting/stopping weave," https://github.com/weaveworks/scope/pull/867, 2021.
[60] S. Lewis, "Qualitative inquiry and research design: Choosing among five approaches," *Health promotion practice*, vol. 16, no. 4, pp. 473–475, 2015.
[61] R. Pike, "Go at google," in *OOPSLA*, 2012, p. 5–6.
[62] J. Meyerson, "The go programming language," *IEEE Software*, vol. 31, no. 5, pp. 104–104, 2014.
[63] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *ISSRE*, 2020, pp. 403–413.
[64] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in github," *Communications of the ACM*, vol. 60, no. 10, p. 91–100, Sep. 2017.
[65] "https://github.com/covidwatchorg/portal/pull/268," https://github.com/covidwatchorg/portal/pull/268, 2021.
[66] H. Sutter and J. Larus, "Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry." *Queue*, vol. 3, no. 7, p. 54–62, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1095408.1095421
[67] "Awaitility library," 2020, http://www.awaitility.org.
[68] "Handle race condition (file deleted after directory scan) that was causing flaky test," https://github.com/eobrain/bajel/issues/33, 2021.
[69] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *MSR*, 2014, p. 82–91.
[70] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *ICSME*, 2018.
[71] W. Wang, W. Lam, and T. Xie, *An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools*, 2021, p. 165–176.
[72] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *ICSME*, 2015, p. 101–110.
[73] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of practitioners' perceptions of test flakiness factors," *STVR*, vol. 1, no. 8, p. e1791, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1791
[74] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.
[75] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *IEEE International Conference on Software Testing*, 2021. [Online]. Available: https://arxiv.org/abs/2101.09077
[76] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019, pp. 312–322.
[77] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *ICST*, 2018, pp. 1–11.
[78] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *MSR*, 2021.
[79] M. Fowler, "Continuous integration," https://www.martinfowler.com/articles/continuousIntegration.html, 2021.