A Study on the Lifecycle of Flaky Tests

Wing Lam
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
winglam2@illinois.edu

Kıvanç Muşlu
Microsoft
Redmond, Washington, USA
kivanc.muslu@microsoft.com

Hitesh Sajnani
Microsoft
Redmond, Washington, USA
hitsaj@microsoft.com

Suresh Thummalapenta
Microsoft
Redmond, Washington, USA
suthumma@microsoft.com

ABSTRACT

During regression testing, developers rely on the pass or fail outcomes of tests to check whether changes broke existing functionality. Thus, *flaky tests*, which nondeterministically pass or fail on the same code, are problematic because they provide misleading signals during regression testing. Although flaky tests are the focus of several existing studies, none of them study (1) the reoccurrence, runtimes, and time-before-fix of flaky tests, and (2) flaky tests in-depth on proprietary projects.

This paper fills this knowledge gap about flaky tests and investigates whether prior categorization work on flaky tests also apply to proprietary projects. Specifically, we study the lifecycle of flaky tests in six large-scale proprietary projects at Microsoft. We find, as in prior work, that asynchronous calls are the leading cause of flaky tests in these Microsoft projects. Therefore, we propose the first automated solution, called *Flakiness and Time Balancer (FaTB)*, to reduce the frequency of flaky-test failures caused by asynchronous calls. Our evaluation of five such flaky tests shows that FaTB can reduce the running times of these tests by up to 78% without empirically affecting the frequency of their flaky-test failures. Lastly, our study finds several cases where developers claim they "fixed" a flaky test but our empirical experiments show that their changes do not fix or reduce these tests' frequency of flaky-test failures. Future studies should be more cautious when basing their results on changes that developers claim to be "fixes".

CCS CONCEPTS

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

KEYWORDS

flaky test, empirical study, lifecycle

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3381749

ACM Reference Format:
Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3381749

1 INTRODUCTION

Developers typically rely on regression testing to ensure that their recent changes do not introduce faults. Ideally, test failures during regression testing would reliably signal issues with the developers' recent changes. Unfortunately, tests that pass and fail nondeterministically on the same code prohibit this ideal. These tests are commonly called *flaky tests* [10, 20], and they negatively impact developers by providing misleading signals about the developers' recent changes.

Flaky tests are a major problem in both industry and research. In recent years, several companies have reported the impact flaky tests have on them. Micco [23] reported that 1.5% of all test runs at Google are flaky, and that almost 16% of their 4.2 million individual tests fail independently of changes in code or tests. Similarly, our previous work [18] reported that about 4.6% of the tests in five Microsoft projects are flaky. At Facebook, Harman and O'Hearn [14] even proposed to adopt the position that *all* tests are flaky (ATAF) and that researchers should rethink testing techniques knowing that they will be used in an ATAF world.

There are a few studies of flaky tests that focus on common categories of flaky-test fixes [20, 25], approaches for detecting flaky tests [11, 13, 19, 27, 31], and automatic approaches for fixing flaky tests [25, 28]. Although this existing work helped substantially advance the topic of flaky tests, none of them study (1) the reoccurrence, execution time (or *runtime* for short), and time-before-fix of flaky tests, which can be relevant to the impact and possible solutions of flaky tests, and (2) flaky tests in-depth on proprietary projects. Studying flaky tests in-depth specifically on the categorization of them on proprietary projects can be important to understand whether new or existing techniques for dealing with flaky tests would work well for both proprietary and open-source projects.

To fill this knowledge gap, we study the lifecycle of flaky tests on six large-scale, diverse proprietary projects at Microsoft. Specifically, we study the overall lifecycle—prevalence, reproducibility, characteristics (e.g., reoccurrence, runtimes), categories, and resolution (e.g., time-before-fix) of flaky tests. Our study of prevalence and reproducibility reveals the substantial negative impact that flaky tests have on developers at Microsoft, while our study on the

characteristics, categories, and resolution of flaky tests confirms that some of the findings from studies on open-source projects also hold for proprietary projects. For example, similar to two prior studies [20, 25] on flaky tests in open-source projects, we also find that the most common category of flaky tests in proprietary projects is the Async Wait category. Tests in this category are flaky because they make asynchronous calls without properly waiting for the call to return. Realizing the substantial impact that flaky tests have on developers at Microsoft and how common Async Wait flaky tests (or *Async Wait tests* for short) are, we propose an automated solution to alleviate the negative impact of these tests.

Our automated solution, called the *Flakiness and Time Balancer* (*FaTB*), alleviates the negative impact of Async Wait tests. Specifically, FaTB identifies the method calls in the test code that are related to timeouts or thread waits, and then calculates the frequency that the flaky test will fail (or *flaky-test-failure rate* for short). Based on the current flaky-test-failure rate, FaTB will then try various time values and outputs the minimum time values that developers should use depending on their tolerance for flaky-test failures. Our evaluation of FaTB on five versions each of five flaky tests shows that the tests can run up to 78% faster and still achieve the same flaky-test-failure rate as before. More importantly, we also find that for some tests, the developers thought that they "fixed" the flaky tests by increasing some time values in the tests, but our empirical experiments show that these time values actually have no effect on the tests' flaky-test-failure rates. Our finding suggests that what developers claim as "fixes" for flaky tests in bug reports, commit messages, etc. can be unreliable. Although, some existing flaky-test studies [20, 25] relied on these claims from developers, future work should be more cautious when basing their results on changes that developers claim to be "fixes".

In summary, we present the following main contributions.

(1) We confirm whether the categorization findings on flaky tests of prior studies [20, 25] conducted on open-source projects are true for the 6 proprietary projects at Microsoft.
(2) We study the lifecycle of flaky tests, which helps us understand them and demonstrates the need for our automated solution. As part of our study, we are the first to investigate the reoccurrence, runtimes, and time-before-fix of flaky tests. The data we use for our study is available online [4].
(3) We propose an automated solution, called FaTB, to balance test flakiness and runtime. Our empirical experiments show that FaTB can help tests run up to 78% faster and the tests will still have the same flaky-test-failure rates as before.
(4) We show that categorization based on changes developers claim as "fixes" can be unreliable. Our finding invalidates a main assumption of prior studies [20, 25], and reveals the need for future work to more cautiously use such "fixes" and to confirm flaky-test fixes.

2 BACKGROUND

Microsoft uses a modern build and test service framework on the cloud, called *CloudBuild* [9]. CloudBuild is an incremental and distributed system for building code and executing tests, similar to other engineering systems such as Bazel [2] and Buck [3]. When CloudBuild receives a build request with a change, it identifies all modules that are impacted by the change. CloudBuild executes the tests only in those impacted modules, and skips the remaining modules' tests, since none of their dependencies changed. Note that, within a module, CloudBuild always executes all tests in the same order (sorted alphabetically).

To address the misleading signals of flaky tests, CloudBuild offers a comprehensive flaky test management system called *Flakes*, that includes four major features: Detection, Reporting, Suppression, and Resolution. The *Detection* feature aims at inferring flaky tests among all tests executed by CloudBuild. More specifically, whenever there is a test failure, CloudBuild automatically retries the test once by default, and if the retry passes, then the test is considered flaky and the build continues. Once a test is considered flaky, Flakes proceeds to *Reporting* where it reports the flaky test to developers by automatically creating a bug report. These bug reports help notify developers of the flaky tests and encourages the developers to fix the flaky tests. Note that Flakes will link multiple flaky tests to the same bug report by looking for similarities in the flaky tests' error messages. Doing so prevents tests with the same root cause from creating many different bug reports.

For *Suppression*, Flakes updates a suppression file that maintains all known flaky tests within the project. Specifically, Flakes adds information (e.g., error message, bug report URL, code version, fully-qualified test name) about a test to the suppression file when a test is found to be flaky. This suppression file is primarily used to suppress future failures of flaky tests, since they are known to be flaky already. Flakes can suppress these failures to help reduce developers' effort in diagnosing test failures due to flaky tests. However, to discourage developers from fully relying on suppressions to deal with flaky tests, Flakes simply suppresses the failures for 30 days by default. Finally, for *Resolution*, when developers close a bug report related to a flaky test, Flakes automatically removes the test from the suppression file. If the test is found to be flaky later, Flakes will reopen a new bug report and repeat all of the steps above. Today, Flakes is used by 11 projects in total at Microsoft. Of these 11 projects, Flakes has already found at least one flaky test in six of the projects. Across all of these projects, Flakes has created over 4,000 bug reports and between May to August 2019, Flakes suppressed over 218,000 flaky-test failures for these six projects.

3 STUDY SETUP

This section describes the projects and datasets of flaky tests we use in our study, the research questions of our study, and how we use the datasets for each question.

3.1 Evaluation projects

Table 1 provides some statistics collected during July 2019, over a 30-day period, for six projects that use Flakes. Each of these projects has at least one flaky test in it currently or had one sometime in its past. Due to company confidentiality reasons, the names of the projects are anonymized. None of the authors has worked on any of the projects that uses Flakes. In the table, Column 2 shows the number of distinct tests in each project. Column 3 shows the number of failed builds for each project. Column 4 shows the number of tests executed in all builds. Note that CloudBuild does not execute all tests in each build; rather, it executes only those tests that are

Table 1: Statistics of the projects with flaky tests using Flakes during July 2019 (over a 30-day period).

Project	# Tests	# Failed Builds	# Test executions	Median build time (min)	# Flaky-test failures	# Builds with flaky-test failures	Project purpose
ProjA	7,281	2,127	2,045,513	29	157	68 (3.2%)	Ads
ProjB	29,589	13,025	45,063,356	21	17,064	1,133 (8.7%)	Cloud computing
ProjC	2,866	2,047	1,429,295	3	24	22 (1.1%)	Engr. infrastructure
ProjD	3,182	9,371	28,557,302	10	39	35 (0.4%)	Database
ProjE	7,939	847	4,702,325	11	734	302 (35.7%)	Engr. Monitoring
ProjF	4,197	491	332,557	27	1,775	133 (27.1%)	Search

Table 2: Flaky-test statistics of the projects in our study. *ProjB's pull requests (PRs) are inaccessible for our study.

Project	# Flaky tests	# Fixed flaky tests	# Flaky test w/ PRs
ProjA	10	3	2
ProjB	352	31	*0
ProjC	73	63	8
ProjD	1453	878	96
ProjE	176	64	27
ProjF	25	1	1
Total	**2089**	**1040**	**134**

within the modules impacted by the change. Column 5 presents the median time of each build in minutes, and Column 6 shows the number of total flaky-test failures suppressed by Flakes. Column 7 shows the number and percentage of failed builds that contained at least one flaky-test failure suppressed by Flakes. Note that each of these builds can have more than one flaky-test failure. Finally, Column 8 shows the purpose of the project. As this table shows, the projects that use Flakes and have at least one flaky test are quite diverse. Specifically, the median build times for these projects vary from 3 to 29 minutes and they all have distinct purposes.

3.2 Datasets

We conduct our study of flaky tests at Microsoft using three datasets. Figure 1 shows an overview for how we obtain these three datasets from the six projects that use Flakes. As Figure 1 shows, we obtain the datasets using three main steps, with each subsequent step using some or all of the data in the previous step. To obtain our datasets, we start with all versions of the suppresion files that Flakes maintains for each of the six projects. These suppresion files are version-controlled, and Flakes uses them to keep track of known flaky tests. Having previous versions of these suppresion files consequently allows us to find flaky tests that Flakes found in the past regardless of whether these tests are fixed or not. In total, Flakes identified 2089 flaky tests from the entire history of the six projects shown in Table 1.

We use the suppresion files maintained by Flakes for each project to create three datasets labeled as All-Fixed, Pull-Requests, and Categorized. Dataset *All-Fixed* includes all flaky tests that Flakes has observed to be fixed and contains 1040 flaky tests. Dataset *Pull-Requests* includes all flaky tests that are fixed and the bug report associated with the flaky test includes a pull request that the developer manually linked to the bug report. This dataset contains 134 flaky tests. Lastly, dataset *Categorized* includes all flaky tests that have pull requests and, upon our manual investigation of the pull requests, bug reports, and source and test code, we categorize these flaky tests with the categories defined in prior studies [20, 25]. This dataset also contains 134 flaky tests.

To obtain the All-Fixed dataset (the result of Step 1), we parse the suppresion files of Flakes into a SQL-like database known as Azure Data Explorer [1]. We parse the suppresion files from the oldest to the newest version, and when we see a flaky test get added to the file, we consider that test to be flaky. On the other hand, when we see a flaky test get removed from the file, we consider that test to be fixed.[1] The number of times a flaky test is detected and fixed depends on the number of times the test is added and removed (respectively) from the suppresion file.

To obtain the Pull-Requests dataset (the result of Step 2), we join the All-Fixed dataset with an existing Azure Data Explorer table that keeps track of which pull requests, if any, are linked to a bug report. Not all closed bug reports are linked to a pull request, because developers have to manually link the pull requests themselves. We join the All-Fixed dataset with an existing Azure Data Explorer table because Flakes keeps track only of the bug report it creates for a particular flaky test, and would not otherwise know if a particular bug report has pull request(s) linked to it. Note that by design ProjB's bug reports are not accessible through Azure Data Explorer. Therefore, all bug report and pull request information for the flaky tests of ProjB is omitted from our study.

To obtain the Categorized dataset (the result of Step 3), we study the pull request, source code, and test code of each flaky test in the Pull-Requests dataset. Each flaky test that we categorize is verified by two or more of the authors independently. Our categorization considers four kinds of locations and 12 root-cause categories that we obtain from two prior studies [20, 25] on flaky tests.

Table 2 summarizes for each project the flaky tests we find in it. Overall, Flakes identified 2089 flaky tests from six projects. On average, Flakes has been tracking flaky test information in these six projects for 181 days. Of the 2089 flaky tests, 1040 tests are fixed. Of the 1040 flaky tests that are fixed, we find that the developers attached a pull request to the bug report for 134 of the tests.

3.3 Research questions

To better understand the lifecycle of flaky tests at Microsoft, we study the prevalence, reproducibility, characteristics, categories, and resolution of flaky tests. More specifically, we address the following research questions:

[1] A flaky test can also be removed from the suppresion files because the test is removed from the project. Our All-Fixed dataset does include such tests.

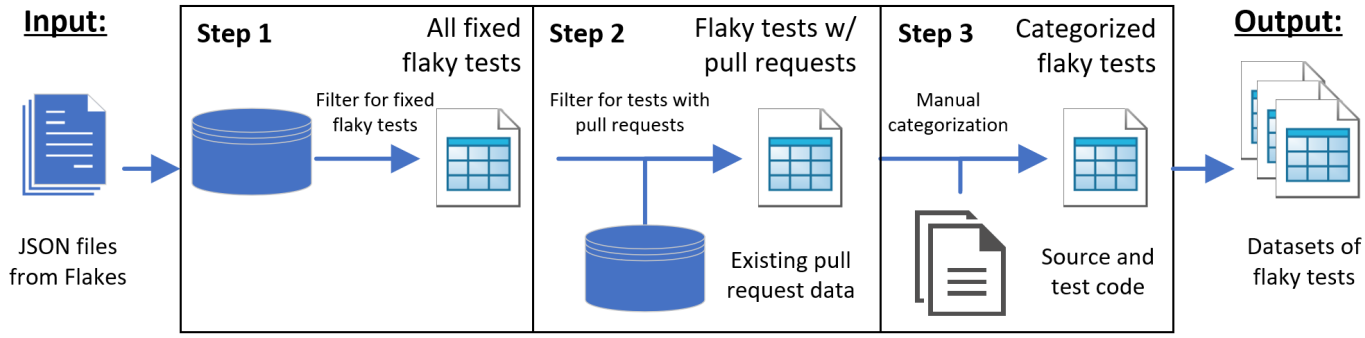

Figure 1: Overview of how we obtain the datasets used in our study.

RQ1 [Prevalence]: How prevalent are flaky tests and to what extent do they impact developers' workflow?
RQ2 [Reproducibility]: How many runs are needed to reproduce flaky-test failures?
RQ3 [Characteristics]: Does test flakiness reoccur after fixes? If so, what are the reasons for it to reoccur?
RQ4 [Characteristics]: How does the runtime of a flaky test differ between passing and failing runs?
RQ5 [Categories]: What are the categories (e.g., root cause, location) of the flaky-test fixes?
RQ6 [Resolution]: How much time do developers take to fix flaky tests?
RQ7 [Resolution]: How effective are developers at identifying and fixing the timing-related Async Wait issues in flaky tests?

We first address RQ1 to understand how problematic flaky tests are at Microsoft. We then address RQ2 to understand the difficulty developers may have in debugging and fixing flaky tests. Knowing the difficulty of reproducing flaky-test failures, we then address RQ3 and RQ4 to understand the characteristics of these flaky tests. We then address RQ5 to extend our characteristics study by categorizing the location and root cause of the flaky-test fixes in our dataset. Lastly, we address RQ6 and RQ7 to understand how effective developers are at fixing flaky tests.

3.4 Methodology

All of our research questions use either the All-Fixed, Pull-Requests, or Categorized datasets of flaky tests described in Section 3.2.

3.4.1 RQ1: Prevalence and impact of flaky tests. For RQ1, we use the All-Fixed dataset, our entire dataset of fixed flaky tests. For this RQ, we rely on the information collected by Microsoft's Flakes during July 2019. Specifically, we look at the number of failed builds that would have occurred due to flaky-test failures if Flakes did not suppress such failures from these builds.

3.4.2 RQ2 and RQ4: Reproducibility and runtime of flaky tests. For RQ2 and RQ4, we again start by using the All-Fixed dataset. Specifically, for each flaky test, we run the test 500 times in the actual build and testing environment. We use only three projects (ProjC, ProjD, ProjE), because we perform these experiments on real proprietary projects, and we cannot interrupt or slow the actual testing environments of the other projects.

3.4.3 RQ3: Reoccurrence of flaky tests. For RQ3, we use the All-Fixed dataset, but we filter for tests that have been fixed more than once. We identify the flaky tests that are fixed more than once by looking for tests that are removed from a project's suppresion file more than once. For the tests that are fixed more than once, we look at the commits, bug reports, and source and test code to understand why they reoccur. We use the commits instead of pull requests because not all tests in the All-Fixed dataset have pull requests linked to the tests' bug reports. Indeed, we find that for the flaky tests that we study for this RQ, all of their bug reports do not have pull requests. We obtain the commits for these tests, by using the dates of their bug reports and the version-control history of the test code to find the likely commits for these tests. We then confirm these commits with the developers of the flaky tests.

3.4.4 RQ5: Categories of flaky-test fixes. For RQ5, we use the Pull-Requests dataset, which contains 134 flaky tests that are fixed and have a pull request associated with the test. Specifically, we study (1) where are the changes located in (i.e., source or test code), and (2) what root causes of flaky-test fixes do these pull requests belong to? Prior studies [20, 25] on flaky tests have found four kinds of locations in which fixes were located in and also identified 12 root causes of flaky-test fixes. For our study, we manually label each pull request along with its corresponding bug report and source/test code with the same four kinds of locations and 12 root causes as the prior studies. We decide to use pull requests, which consists of one or more commits, because pull requests represent a more complete set of changes. These changes from pull requests generally build without errors and have been tested on the developers' machines to ensure that they do not fail any tests.

3.4.5 RQ6: Time-before-fix of flaky tests. For RQ6, we use the All-Fixed dataset. Specifically, for each fixed test, we study the bug report linked to the test. Recall that Flakes' Reporting feature, as described in Section 2, will automatically create a bug report for each test it finds to be flaky. To obtain the time-before-fix of flaky tests, we study the time the bug reports of these tests took from being created to them being closed.

3.4.6 RQ7: Developers' effectiveness on identifying and fixing Async Wait tests. For RQ7, we use the Categorized dataset, which contains 134 flaky tests that are fixed, have a pull request associated with the test, and its pull request, bug report, and code is categorized.

Since in RQ5 we find that Async Wait is the most common category of flaky tests, we focus specifically on this category for RQ7. To understand how effective developers are at identifying and fixing Async Wait tests, we sample five Async Wait tests whose fix by developers is to increase the wait/timeout. We then calculate the flaky-test-failure rate with the developer-suggested fix and measure how the rate changes when the time value increases or decreases.

4 ANALYSIS OF THE RESULTS

This section presents the results of our study for the research questions in Section 3.3 using the methodology we describe in Section 3.4. The data we use for our study is available online [4].

4.1 RQ1: Prevalence and impact of flaky tests

We begin our study by first investigating how prevalent flaky tests are at Microsoft. From Tables 1 and 2, we see that for all projects except for ProjD, the number of flaky tests ever found is only a small fraction of the total number of tests these projects' have during the month of July 2019. However, just because a project contains many or few flaky tests it does not necessarily mean that the developers' workflow are often or rarely impacted by these tests. To understand whether these flaky tests do impact developers' workflow or not, we also show in Table 1 the percentage of developers' builds in which the build would have failed due to flaky-test failures if Flakes did not suppress such failures. One interesting thing to note here is that even though some projects have lots of flaky tests, these projects' chance for builds to fail due to flaky-test failures are not particularly high. For example, ProjD has 1453 flaky tests with over half still not fixed, but flaky-test failures only affects 0.4% of its builds over a 30-day period, while ProjE has only 176 flaky tests but its builds are affected by flaky-test failures 35.7% of the time during the same period. Our results demonstrate that, although flaky tests may not always be very prevalent, the percentage of builds that are impacted by flaky-test failures can still be quite substantial.

4.2 RQ2: Reproducibility of flaky-test failures

One of the biggest challenges developers have when debugging or fixing flaky tests is to reproduce the flaky-test failure. To understand how much of an imposition the reproducibility of flaky-test failures may have on developers, we study the number of flaky tests in which we can reproduce the flaky-test failures, and for the tests where we can reproduce the flaky-test failure, we also study these tests' flaky-test-failure rates. For each of the flaky tests that we use for this RQ, we run the test 500 times using the same configuration of the machines and the version of code that Flakes detected the test to be flaky on.

Table 3 summarizes our results. We use only a subset of our dataset for this RQ, because these experiments are performed on real proprietary projects, and we could not slow down the actual testing environment of the other projects. Furthermore, not all flaky tests can be run again due to problems compiling the version of code that the test was found to be flaky on. The actual number of flaky tests that we are able to run 500 times for is shown in Column 2. Column 3 shows the number of tests that pass and fail at least once, and Columns 4 and 5 show the average and median

Table 3: Statistics on reproducibility of flaky-test failures.

Proj.	# Flaky tests	# Flaky tests 1+ pass & fail	Average % fail	Median % fail
ProjC	7	3 (43%)	0.2	0.2
ProjD	545	95 (17%)	36.8	29.4
ProjE	85	21 (25%)	9.7	0.6

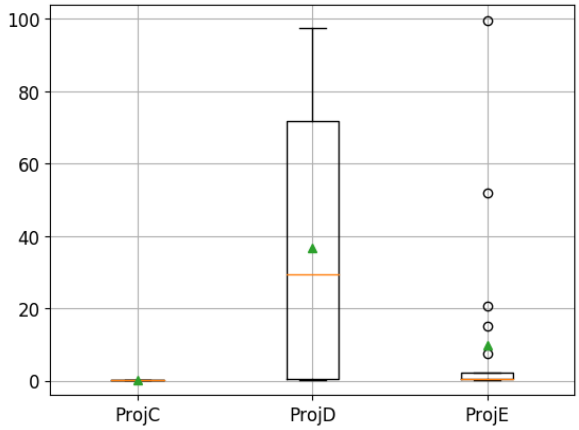

Figure 2: Percentage rate of flaky-test failures.

(respectively) percentage rate of flaky-test failures for the flaky tests that pass and fail at least once.

Column 2 of the Table 3 shows that flaky-test failures are reproducible between 25% to 43% depending on the project. This finding suggests that there are many flaky tests (up to 75% depending on the project) where even with 500 runs, we cannot reproduce the flaky-test failures of these tests. We also see from Column 4 that the median percentage rate of flaky-test failures can be quite low, particularly for ProjC and ProjE. This finding suggests that even when flaky-test failures can be reproduced in 500 runs, only a small number of failures are reproduced.

Figure 2 shows a box plot for the percentage rate of flaky-test failures for the flaky tests that pass and fail at least once in each project. We can see in this figure that the averages of both ProjD and ProjE (Column 3 in Table 3) is quite high due to the percentage rates of 23 outlier tests. To understand these outliers better, we analyze them and find that 9 of these tests are likely Async Wait—their names contain "Async". Of these 9 tests, we have the pull request, bug report, and code for 3 of the tests, and while studying the categorization of flaky tests in RQ5 (Section 4.5), we do indeed categorize these 3 tests as Async Wait tests.

4.3 RQ3: Reoccurrence of test flakiness

As Section 4.2 demonstrates, flaky-test failures can be quite difficult for developers to reproduce. This difficulty makes it so that when developers are fixing flaky tests, they may just assume that their changes fixed the flaky-test failure and they may not actually confirm their assumption. For our study on the reoccurrence of flaky tests, we use the All-Fixed dataset, which contains 1040 fixed flaky tests. Of these 1040 flaky tests, we find that four flaky tests are fixed more than once.

If a flaky test is found to be flaky more than once, then it is either because (1) the developers' initial fix for the flakiness was inadequate, or (2) the cause of flakiness was reintroduced. In either case, sufficient time must be given for the developers to notice the reoccurrence and for the test to run many times for it to fail builds again. On average, the fixed flaky tests in our study have been fixed for more than 86 days.

To understand why four flaky tests had to be fixed more than once, we manually investigate the the commits, bug reports, and the source and test code for each flaky test. We use commits instead of pull requests because not all tests in the All-Fixed dataset have pull requests, and all four tests that are fixed more than once do not have pull requests. We obtain the commits for these tests by using the time their bug reports closed and the version-control history of the test code to find the likely commits for these tests. We then confirm the likely commits with the developers of the four tests.

Our investigation into the four flaky tests that reoccurs more than once reveals that all four reoccurrences are due to case (1) when the developers' initial fix was inadequate. For these flaky tests, we can clearly see this being the case since the developers described their initial fix as being inadequate in their latest fix. For example, one developer described his latest fix as "Increase the wait time for idle timeout test case to ensure that the receive loop exits first. Previously, the wait time was 1 second more than the idle timeout, which was cutting it too fine". Overall, our investigation into the reoccurrence of these four tests finds that developers have the following important sentiments about fixing flaky tests.

(1) Developers are rarely able to reproduce the flaky-test failures locally on their own machines or on servers.
(2) Consequent of (1), developers often resort to making multiple changes to fix test flakiness. These changes are often either
   (a) made by trial-and-error guessing and the developers rely on the frequent runs of the test on the servers to determine whether the fix was adequate, or
   (b) made simply to log additional information so that the developers can know more about the flaky-test failure before attempting a real fix.

4.4 RQ4: Runtime of flaky tests

To begin understanding why a test may be flaky, we study the runtime of flaky tests. We study the runtime of flaky tests because one may think that when flaky tests fail they would run faster than their passing runs, since the test may have encountered a fault and stopped early. However, flaky tests may also take longer in their failing runs if they are flaky because they time out. In such cases, the flaky test may wait for a callback that simply never happens, indicating that these tests likely make asynchronous calls. Similar to RQ2 (Section 4.2), we can use only a subset of our dataset for this RQ since we could not slowdown the testing environment of the other projects, and the version of code in which some flaky tests were detected on no longer compiles.

Table 4 and Figure 3 shows the runtime in seconds of the flaky tests that pass and fail at least once in 500 runs. Overall, we see that for ProjE, the average and median runtime of passing runs is more than failing runs. As for ProjC, we see that the average and median runtime of passing runs is about the same as the failing runs. This result suggests that for these two projects, their flaky tests are likely unrelated to asynchronous method calls. On the other hand, we can see that for ProjD, the average and median runtime of failing runs is substantially more than the runtime of passing runs. This result suggests that ProjD's flaky tests are likely related to asynchronous method calls. When we categorize flaky tests in RQ5, we do indeed find that the majority of ProjD's flaky tests are Async Wait tests.

Table 4: Statistics on runtime (in seconds) of flaky tests.

Proj.	Test result	# Runs	Average runtime	Median runtime
ProjC	Pass	1,497	2.39	1.34
ProjC	Fail	3	2.47	1.48
ProjD	Pass	30,023	9.41	4.31
ProjD	Fail	17,477	22.25	25.00
ProjE	Pass	9,477	2.14	1.61
ProjE	Fail	1,023	1.72	0.80

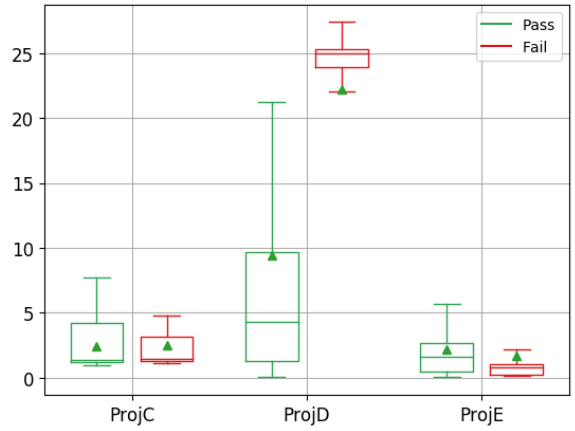

Figure 3: Runtime (in seconds) of flaky tests.

4.5 RQ5: Categories of flaky-test fixes

To understand the categories of flaky-test fixes, we use the Pull-Requests dataset, which contains 134 fixed flaky tests that all have an associated pull request. We categorize these tests by studying their pull requests, bug reports, and source and test code. Our study focuses on two main questions; (1) where were the majority of the changes located in (i.e., source or test code), and (2) what root causes of flaky-test fixes do these pull requests belong to? Prior studies [20, 25] on flaky tests found four kinds of location in which fixes were located and also identified 12 root causes of flaky-test fixes. Table 5 summarizes our findings and those of prior studies.

4.5.1 Location of flaky-test fixes. A prior study [20] on flaky tests identified four kinds of locations for flaky-test fixes; (1) Source code only, (2) Test code only, (3) Source and Test code, and (4) Configuration. We adopt similar kinds of locations for our study. The only change we make is that "Configuration" is changed to "Other" instead, and a fix is considered to be "Other" if it includes changing anything besides source or test code (e.g., test input data, configuration).

Table 5: Comparison of flaky-test-fix categories with previous studies [20, 25]. "-" denotes that such data was not made available in their paper or otherwise.

Categories	Our study	[20]	[25]
Location of fixes			
Source only	1%	12%	-
Test only	71%	73%	>=61%
Test and Source	11%	11%	-
Other	17%	4%	-
Root cause of fixes			
Async Wait	78%	45%	27%
Network	14%	6%	10%
Concurrency	8%	20%	17%
Resource Leak	5%	7%	3%
Randomness	5%	2%	3%
IO	5%	2%	22%
Time	4%	3%	3%
Floating Point Operations	2%	2%	2%
Test Order Dependency	0%	12%	12%
Unordered Collections	0%	1%	1%
Difficult to categorize	26%	20%	-

Another flaky test study [25] investigated the categories of flaky tests and how it relates to code smells. They did not present their findings on how many fixes for flaky tests are in source code or other files. Nevertheless, they found that at least 61% of flaky tests could be fixed by manually fixing three code smells in the tests.

Overall, our findings confirm what prior studies [20, 25] found: the majority of fixes (71%) for flaky tests are in the test code. Interestingly, we find that, in 5% of the fixes, developers simply removed the test. Our investigation shows that developers sometimes temporarily removed the failing test or claimed that the test is for functionality that is no longer supported. We also find that about 12% (1% + 11%) of fixes involve changes to source code. Overall, our results show that ignoring flaky tests can be dangerous since they do indicate faults in both source and test code.

4.5.2 Root causes of flaky-test fixes. When performing our study on the categories of flaky-test fixes, we use the same categories as the prior studies. We find that the most common category of fixes is Async Wait, with 78% of the fixes belonging to that category. Async Wait flaky tests make an asynchronous call and they do not properly wait for the call to return. The second most common category is Network with 14% of the fixes. Note that unlike the prior studies, one fix of ours may belong to multiple categories. Also, similar to a prior study [20], we find a number of fixes (26%) that we could not categorize due to the large number of changes. Specifically, in our study, these fixes modify an average of 785 files. When we examine such fixes in detail, we see that they were a part of a version upgrade or major refactoring.

Overall, our study differs from prior studies [20, 25] in two main ways. (1) Both prior studies used flaky tests from open-source projects, while we used flaky tests from proprietary projects at Microsoft, and (2) we study the pull requests, bug reports, and source and test code of flaky tests, while one prior study [20] studied only commits and another prior study [25] studied only the test code in one version of many projects. We believe these differences between our studies is responsible for the minor differences in our findings. One example of how our results differ from prior studies is the percentage of fixes that are categorized as Test Order Dependency. This difference is likely because the way we run tests at Microsoft heavily reduces the chance of Test Order Dependency causing test flakiness. As explained in Section 2, CloudBuild always runs tests in the same order, but this requirement is not true for the open-source projects in prior studies[2]. Indeed, as Table 5 shows, none of the flaky tests within the six projects we study are flaky due to Test Order Dependency, even though this category is the third most common category of flaky tests in open-source projects.

Even though the composition of our study differs from prior studies, our findings on the location and common root causes of fixes remain largely the same. Specifically, we all find that the majority of flaky-test fixes are located in test code, but a nontrivial amount of them do also involve source code (12%). Also, the most common category of flaky-test fixes is Async Wait. Our findings here suggest that solutions, like the one we propose in Section 4.7, that can help reduce flaky-test failures of Async Wait tests would highly help alleviate the negative impact of flaky tests.

Table 6: Time given in days for developers to close flaky-test or non-flaky-test related bug reports (BRs). ProjB is omitted because its BRs are inaccessible for our study.

	Flaky-test			Non-flaky-test		
Proj.	**# BRs**	**Median**	**Avg.**	**# BRs**	**Median**	**Avg.**
ProjA	2	5	5	238	3	17
ProjC	55	90	95	96	11	30
ProjD	759	6	11	1575	8	12
ProjE	43	14	39	55	10	13
ProjF	1	8	8	3	8	12
Overall	**861**	**7**	**18**	**1967**	**7**	**13**

4.6 RQ6: Time-before-fix of flaky tests

Prior work [7, 14, 15, 18, 23, 26] has highlighted how flaky tests negatively impact developers' software development process and how important it is for developers to fix these flaky tests. To understand whether developers at Microsoft understand the importance of fixing flaky tests, we study how long developers take on average to fix flaky tests. More specifically, we study the time developers take on average to close the bug report linked to a flaky test. At Microsoft closing a bug report typically means that the bug has been fixed. For our study, we start with the flaky tests in our All-Fixed dataset, which consists of 1040 fixed flaky tests. For the 1040 flaky tests in our dataset, we find that these flaky tests are linked to 861 bug reports. As we explain in Section 2, multiple flaky tests may be linked to the same bug report if these flaky tests share similar error messages (e.g., two tests are flaky due to the same setup method).

[2]Note that it is still possible for a flaky test to fail due to Test Order Dependency at Microsoft, because the flaky test could fail when a new test is added and the new test runs before the flaky test. Dually, a flaky test could start failing as well when a test that was needed to run before the flaky test is removed from the test suite.

Table 7: Categorization of Async Wait flaky-test fixes.

Categories	Tests
Timing-related fix	
Increase wait/timeout	21 (31%)
Add/improve callback	19 (28%)
Add/improve polling	7 (10%)
Timing-unrelated fix	
Removing code	17 (25%)
Mocking async calls	10 (15%)
Difficult to categorize	20 (23%)

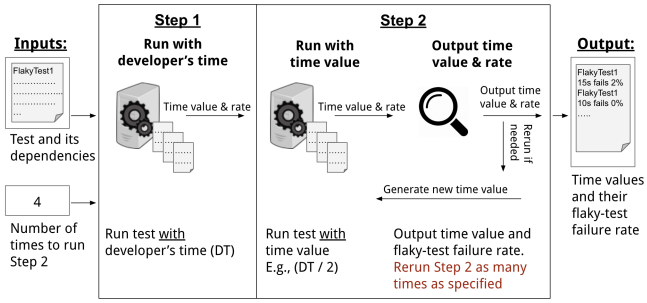

Figure 4: Overview of how the Flakiness and Time Balancer (FaTB) works.

Table 6 shows the average and median number of days each project takes to close flaky-test and non-flaky-test related bug reports. Note that ProjB's results are omitted from Table 6, because as we explain in Section 3.2, all bug report and pull request information for the flaky tests of ProjB are inaccessible for our study. As we show in Table 6, developers take, on average, 18 days, with a median of 7 days to close flaky-test related bug reports, while they take on average 13 days, with a median of 7 days to close all non-flaky-test related bug reports. The median time developers take to close flaky-test and non-flaky-test related bug reports are the same, suggesting that developers consider these bug reports to be of equal importance. However, when we compare the average time developers take to close flaky-test related bug reports to the non-flaky-test ones, we see that flaky-test related ones take substantially longer than non-flaky-test ones (18 days for flaky-test related ones compared to 13 days for non-flaky-test related ones). As Table 6 shows, part of the reason why the average for flaky-test related ones are higher is because the flaky-test related bug reports in ProjC take much longer to close than the non-flaky-test ones. When we compare the flaky-test and non-flaky-test related bug reports per project, we see that ProjA's and ProjF's average time to close non-flaky-test related bug reports are actually more than the time to close flaky-test related ones. On the contrary, we also see that ProjC's and ProjE's average time to close flaky-test related bug reports are substantially more than the time to close non-flaky-test related ones. Our findings suggest that although there has been a substantial amount of work from both industry and academia on flaky tests, it can still be important to communicate to some developers the importance of fixing flaky tests. Following our study, we personally approached a number of teams (i.e., those from ProjC and ProjE) to better communicate to them this importance.

4.7 RQ7: Developers' effectiveness on identifying and fixing Async Wait tests

Based on the prevalence of Async Wait tests as described in RQ5 (Section 4.5) and in prior studies on flaky tests [20, 25], we proceed to study developers' effectiveness in identifying and fixing Async Wait tests at Microsoft. To study this RQ we first categorize the Async Wait related flaky-test fixes in our dataset. In total from our work in Section 4.5, we find that there are 87 flaky tests that have fixes related to asynchronous method calls.

Prior work [10] on asynchronous tests proposed three main ways one should test asynchronous code. (1) Create a "synchronous" interface between tests and asynchronous code, (2) implement callbacks on all asynchronous code, and (3) check, or poll, frequently on whether an asynchronous service is complete. When we study the Async Wait flaky-test fixes (or *Async Wait fixes* for short) in our dataset, we find that there are no cases in which (1) was done by developers. Our results are likely because (1) requires substantial effort from developers to create and maintain such interfaces. On the other hand, we do see developers using both callbacks (28% of Async Wait fixes) and polling (10% of Async Wait fixes) to fix their Async Wait tests. Beyond the three categories laid out in this prior study, we also find three different categories for these Async Wait fixes. Specifically, we find that the most common category of fix (31% of Async Wait fixes) involves simply increasing the wait time or timeout of asynchronous method calls. The other two categories are to simply remove code related to the flaky-test failure (25% of Async Wait fixes) or to mock the asynchronous calls (15% of Async Wait fixes). Lastly, we find that 23% of the Async Wait fixes are difficult to categorize, since they involve changes to asynchronous method calls, but we cannot identify any particular categories for these fixes. Table 7 summarizes the findings from our categorization. Note that the fix for each test may be categorized into one or more categories.

4.7.1 Evaluating and improving developers' Async Wait fixes. With the majority of the Async Wait fixes involve simply increasing the wait time or timeout (*time value* for short) of an asynchronous call, we proceed to study how well these time values set by developers are at reducing flaky-test failures and how these time values affect the runtime of these flaky tests. To evaluate and improve developers' Async Wait fixes, we propose the *Flakiness and Time Balancer* (*FaTB*). FaTB first finds the flaky-test-failure rate of the test using the developers' fix. Once FaTB obtains the rate associated with the developers' fix, it then increases or decreases the time value and again measures the flaky-test-failure rate associated with the new time value. Figure 4 shows an overview for what FaTB does.

Depending on whether the test is still flaky with the new time value, FaTB will use that information to either increase or decrease the next time value to try. On a high-level, to lower the time value, FaTB will first use the time value between the developer's fix time and the time set before the fix. For an example, if the developer's

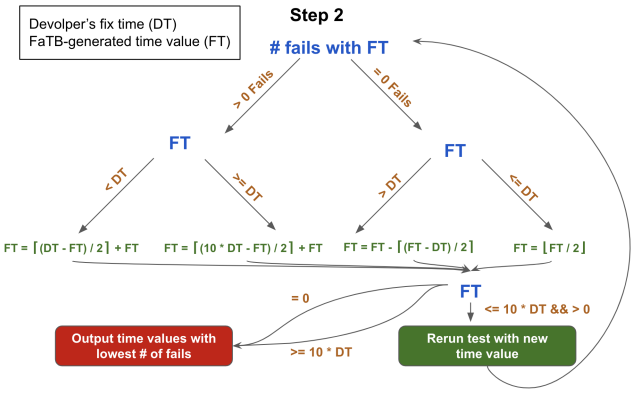

Figure 5: How FaTB chooses the next time value an Async Wait test should try.

fix time was 1000 milliseconds (ms) and the time value before their fix was 500 ms, then FaTB would first set the time value to be 750 ms. If lowering the time value does not cause flaky-test failures in some number of runs (100 by default and for our experiments), then FaTB will take the time between our current time value and zero (e.g., 375 ms). If lowering the time value does cause flaky-test failures, then the next time value will be between the current value and the developer's fix time (e.g., 875 ms).

FaTB outputs the observed flaky-test-failure rate and average test runtime for each time value (e.g., [time value: 375ms, fails: 1%, runtime: 875ms], [time value: 1000ms, fails 0%, runtime: 1500ms]). As a post-processing step, FaTB will also remove time values that have the same flaky-test-failure rate, choosing to output only the minimum time value and runtime for all observed flaky-test-failure rates. This output enables developers to finely balance the trade-off of their tests' runtime and flaky-test-failure rate. The logic FaTB uses to generate different time values is shown in Figure 5. FaTB generates time values specific to the machine on which FaTB is run on. To ensure that these generated time values perform well on different machines, developers can run small benchmarks on these different machines (e.g., their own development machine) and the machines on which they run FaTB (e.g., a development server). The difference in the machines' performance on the small benchmarks can then be used to scale the generated time values as needed.

4.7.2 Results. To evaluate FaTB, we randomly sample five tests from the 21 tests whose fix was to increase the wait/timeout. The results from us applying FaTB on these five flaky tests are shown in Table 8. Specifically, we use FaTB to generate four time values for each test, and Version 0 represents the value the developers proposed to fix the flaky tests with. We run the test 100 times for each version to measure that version's flaky-test-failure rate. Due to confidentiality reasons, the names of the tests are anonymized.

We apply Step 2 of FaTB four times on five flaky tests and find that for four of the flaky tests, even when the time value is set to be substantially lower than the value set by the developers to fix the test, the tests' flaky-test-failure rates appear to be unaffected. More specifically, for Test2, Test3, Test4, and Test5, we see 0% flaky-test-failure rates even when we substantially decrease the time

Table 8: Statistics on the results produced by FaTB when we apply it to five versions of five Async Wait tests. Units for Time value depend on the test. Runtime is in seconds. Prefix value is the value set before the developer's fix. Version 0's time value is the value set after the developer's fix.

Flaky test	Version	Time value	% Fails	Average runtime	Median runtime
Test1	0	600	0	1.39	1.37
(Pre-	1	450	0	1.22	1.22
fix	2	225	73	1.07	1.08
value:	3	413	0	1.19	1.19
300)	4	206	82	1.10	1.07
Test2	0	1,000	0	1.75	1.75
(Pre-	1	800	0	1.61	1.54
fix	2	400	0	1.14	1.13
value:	3	200	0	0.96	0.94
600)	4	100	0	0.84	0.84
Test3	0	600	0	1.67	1.66
(Pre-	1	300	0	1.36	1.36
fix	2	150	0	1.21	1.21
value:	3	75	0	1.14	1.14
0)	4	37	0	1.09	1.08
Test4	0	100	0	8.08	8.03
(Pre-	1	50	0	7.57	7.49
fix	2	25	0	7.43	7.38
value:	3	12	0	7.39	7.30
0)	4	6	0	7.20	7.10
Test5	0	150	0	0.18	0.18
(Pre-	1	83	0	0.11	0.11
fix	2	41	0	0.07	0.07
value:	3	20	0	0.05	0.05
15)	4	10	0	0.04	0.04

value set by the developers. Our finding here further echos the sentiments that we find about fixing flaky tests in Section 4.3. More specifically, we see that the fix employed by the developers for these flaky tests were likely educated guesses that turn out to be unrelated to the flaky-test failures of the test. This finding here is largely related to how truly understanding the root cause of a flaky test is often very complicated. For these four flaky tests, neither we nor the developers were able to actually determine the root cause for the flaky test, and, consequently, neither we nor they are able to fix the flaky test. Future studies on the root causes of flaky tests should be more cautious when basing their results on the changes by developers.

Besides the four flaky tests that do not encounter any flaky-test failures, we see one example (Test1) that does exhibit flaky-test failures once we lower the wait/timeout value of the test. For Test1, we can see that once the value goes lower than the value before the developer fixed the flaky test (300), we start observing a high flaky-test-failure rate (e.g., when set to 225, we see a 73% flaky-test-failure rate). However, once FaTB sets the value to be 413, we see a 0% flaky-test-failure rate. When we compare the flaky-test-failure rate with the developer's fix (time value for Version 0) to the flaky-test-failure rate with the time value for Version 3, we see that this flaky test can obtain the same flaky-test-failure rate when the time value

is set to 600 or 413. However, the lower time value (413) enables this test's average runtime to be about 14% faster than the higher value (600). Similarly, Test2, Test3, Test4, and Test5 do not encounter any flaky-test failures on all versions, and their average runtime can also be faster by about 52%, 35%, 11%, and 78%, respectively.

It is surprising that the developers of Test2, Test3, Test4, and Test5 would increase the time values of these tests when the values do not appear to empirically affect the tests' flaky-test-failure rate. Our results suggest that there are perhaps some other changes in these pull requests that actually fixes the flaky tests, and that the changes in time values were not intended as the fix. To understand why the developers may have made these time value changes, we study the pull request messages of the fixes. We find that for all four of the flaky tests besides Test4, the messages all say that they are increasing the time values as a fix to the flakiness. For example, Test3's message says "Fix is to wait 2 * X time". On the other hand, Test4's message says "Fix flaky test" and then explains how some refactoring was done to some asynchronous code. From these pull request messages, we see that at least for Test2, Test3, and Test5, these developers are purposefully trying to fix their Async Wait tests by increasing time values. Note that Microsoft does not encourage developers to fix their Async Wait tests by increasing the time value. However, since regulating how developers fix their code would be very costly to do, we do plan to use FaTB to help developers at Microsoft. With the prevalence of Async Wait tests and how developers prefer to fix these tests by increasing the time value, we suspect that there are many other tests whose collective reduction in test runtime can substantially lessen the time developers spend waiting for test results, machine resources needed to run these tests, and amount of flaky-test failures developers debug.

5 THREATS TO VALIDITY

Our work contains many of the common threats typically found in empirical studies. In this section we focus on the issues that are more specific to our study.

Subjects of our study. Our study consists of just six projects at Microsoft, and our findings from studying these six projects may not generalize to other projects or companies. To avoid any bias in the selection of our projects, we include all projects using Flakes in our study. As we describe in Section 2, there are a total of 11 projects using Flakes, and of these 11 projects, the six projects we study are the ones where Flakes found at least one flaky test. The projects we study also greatly vary in activity (e.g., number of builds per month) and in purpose (e.g., Database, Search).

Aside from the projects used in our study, our decision to study pull requests, bug reports, and source and test code to understand the fixes of flaky tests in Section 4.5 may also be a threat. Prior studies [20, 25] on flaky tests have used either the commits or just the test code to understand the characteristics of flaky tests. For our study, we use pull requests, which consists of one or more commits, because we believe that pull requests represent a more complete set of changes made by the developers. Unlike commits, changes from pull requests generally build without errors and have been tested locally to ensure they do not fail any tests. Compared against prior work, one prior study [20] did not run the tests and observed them to be flaky like we do. Another prior study [25] did run the tests, but they run the tests only 10 runs instead of 500 runs like we do, and they did not study changes outside of the test code (i.e., bug reports or source code).

Metrics used in our study. The metrics we use in our study poses a potential threat to our findings and results. For example, we use the time a bug report is opened till when it is closed to understand developers' sense of urgency in fixing flaky tests. In reality, a developer taking a short or long amount of time to fix flaky tests could be an indicator of how easy or difficult the fix was and would be irrelevant to the developer's sense of urgency. The timing we report may also be inaccurate, since different teams work differently and some teams may close bugs as soon as they are fixed, while other teams may only close them at their next team meeting.

Flaky tests used in our study. Flaky tests, by definition, may pass or fail on the same code. To identify the flaky tests used in our study, we rely on Flakes, which simply reruns failing tests once to see whether it would pass on the rerun. Since there are no guarantees that the rerun would pass if the test is indeed flaky, Flakes may potentially contain many false negatives, in which a test that is flaky is undetected. Nevertheless, Flakes contains no false positives, meaning that all flaky tests detected by Flakes must indeed be flaky. Due to this threat, the number of flaky tests we report in our study is simply the minimal number of flaky tests in our projects.

Our findings in regards to the runtime and reproducibility of flaky tests identifies that there are some patterns. However, more runs of the flaky tests may change our findings. To mitigate this threat, we choose to run each test a high number of runs, specifically 500 runs. Running the test 500 runs to identify flaky-test failures is substantially more than a prior study on flaky tests [25], which ran the tests for 10 runs.

Findings from manual inspection. Certain research questions in our study requires us to manually inspect the information of flaky tests. Specifically, for categorizing flaky tests and categorizing Async Wait tests, we minimize the occurrence of miscategorization by having more than one author inspect every pull request, every bug report, and all source and test code, and we discussed our categorizations until everyone agrees.

6 RELATED WORK

Studying flaky tests. Luo et al. [20] performed the first extensive study on flaky tests. They manually investigated 201 commits from 51 open-source projects, finding that the primary causes for flakiness are (1) Async Wait, (2) Concurrency, including atomicity violation, data races and deadlocks, and (3) Test Order Dependency. Palomba and Zaidman [25] conducted a similar study on flaky tests in open-source projects. Using code smell detectors, they also find that Async Wait is the most common category of flaky tests and that 61% of flaky test fixes can be attributed to three code smells specific to tests. In comparison to both of these studies, our study is based on flaky tests at Microsoft, and we study flaky tests using pull requests where each test is confirmed to be flaky by our infrastructure. Although, our setup and datasets differ, we confirm that all three studies have similar findings in that Async Wait is the most common category of flaky tests and that most flaky-test fixes are in the test code. Zhang et al. [31] reported that test suites can suffer from *Test Order Dependency* where test outcomes can be affected

by the order in which the tests are run. Lam et al. [19] conducted similar work as them and found 422 flaky tests in 82 projects. Of their dataset, 50.5% of the flaky tests are due to Test Order Dependency, while the remaining are not Test Order Dependency. In our case, CloudBuild always runs tests of a test suite in the same order, therefore we do not find any Test Order Dependency tests. Thorve et al. [29] found additional root causes for flaky tests when studying flaky-test commits in Android. Similar to Luo et al. [20], their study used commits as opposed to the pull requests we use in our study. Eck et al. [7] studied developers' perception of flaky tests and had developers categorize the patches that developers claim to have fixed flaky tests. Similarly, we conduct a study on the pull requests that developers claim to have fixed flaky tests and find that some of these pull requests do not actually fix flaky tests. As more studies are conducted on the fixes of flaky tests, future work should better explore how changes by developers should be used and how one can confirm whether these changes do indeed fix flaky tests.

Reproducibility of flaky tests. Luo et al. [20]'s study was the first to report that reproducing flaky-test failures is difficult. Since then, there has been numerous reports from others on the likelihood to reproduce flaky-test failures. Lam et al. [18] and Labuschagne et al. [17] found that on average, 27.4% and 12.8%, respectively, of builds would fail because of flaky tests. Gao et al. [12] also observed that it is difficult to reliably reproduce results from tests where a user interacts with a system. Palomba and Zaidman [25] found that 45% of all tests they analyzed were flaky. Compared to our work, they found these tests to be flaky using only 10 runs, while our experiments use 500 runs. Our study on the reproducibility of flaky tests finds that the likelihood to reproduce flaky-test failures can range between 17% to 43% depending on the project. Although studies [26] have shown that ignoring flaky-test failures can lead to more crashes in production code, developers do ignore flaky tests. Thorve et al. [29] examined 77 commits pertaining to flaky tests from 29 Android projects, and they found that 13% of the commits simply skipped or removed flaky tests. When we examine the pull requests in our study, we find that developers removed the test to fix 5% of the flaky tests.

Reducing and removing flaky-test failures. Muşlu et al. [24] proposed a technique to run tests in separate processes, and Bell and Kaiser [5] proposed a technique to run tests in the same process as two different means to remove flaky-test failures from Test Order Dependency flaky tests. Shi et al. [28] proposed iFixFlakies, a tool to automatically fix Test Order Dependency flaky tests. At Microsoft, we rarely have Test Order Dependency flaky tests; instead they are mainly Async Wait tests. Our proposed solution, FaTB helps reduce the flaky-test-failure rate of Async Wait tests. Bell et al. [6] proposed DEFLAKER, a technique that monitors the code coverage of tests in a previous version of code and uses the coverage information to inform developers whether a test failure is due to recent changes in future versions of code. Unlike DEFLAKER, FaTB directly helps developers prevent flaky-test failures altogether. Fowler [10] proposed three main ways in testing asynchronous code; (1) creating a synchronous interface between tests and asynchronous code, (2) implementing callbacks on all asynchronous code, and (3) checking frequently on whether an asynchronous service is complete. The implementation of (1) and (2) requires substantial effort from developers to setup and maintain. The implementation of (2) and (3) both still require the developers to provide some timeout value for the asynchronous call which may never complete. Our proposed solution in RQ7, FaTB, can help the developers systematically derive a timeout value that minimizes the runtime and flaky-test-failure rate. Jagannath et al. [16] proposed IMUnit, a new language that allows developers to specify the execution flow of tests that make asynchronous method calls. Similarly, Elmas et al. [8] proposed CONCURRIT, a scripting language that allows developers to control the scheduling of threads to find or reproduce concurrency bugs. Other work [21, 22, 30] proposed tools that help enforce policies specified by the developers. These policies dictate the scheduling in which threads run, and developers have to manually write these policies. Unlike these prior work, FaTB does not require the developers to provide additional information (e.g., policies in which threads should execute), or write their code differently. Instead, FaTB assists the developers by systematically deriving the time a test should wait for asynchronous calls.

7 CONCLUSION

Flaky tests are a major problem in both industry and research. Although flaky tests are the focus of several existing studies, none of them study (1) the reoccurrence, runtimes, and time-before-fix of flaky tests, and (2) flaky tests in-depth on proprietary projects. To fill this knowledge gap, we study the lifecycle of flaky tests on six large-scale, diverse proprietary projects at Microsoft. Our study of prevalence and reproducibility reveals the substantial negative impact that flaky tests have on developers at Microsoft, while our study on the characteristics, categories, and resolution of flaky tests confirms that some of the findings from studies on open-source projects also hold for proprietary projects. For example, similar to two prior studies on flaky tests in open-source projects, we also find that the most common category of flaky tests in proprietary projects is the Async Wait category. To help alleviate the problem of Async Wait flaky tests, we propose the *Flakiness and Time Balancer* (*FaTB*). FaTB identifies the method calls in the test code that are related to timeouts or thread waits, and then it calculates the flaky-test-failure rate of the flaky test. Based on the current flaky-test-failure rate, FaTB then tries various time values and outputs the minimum time values that developers should use depending on their tolerance for flaky-test failures. Our evaluation of FaTB on five versions each of five flaky tests shows that tests can run up to 78% faster and still achieve the same flaky-test-failure rate as before. We also find that the developers thought they "fixed" the flaky tests by increasing some time values in them, but our empirical experiments show that these time values actually have no effect on the flaky-test-failure rates. Our finding suggests that what developers claim as "fixes" for flaky tests in bug reports, commit messages, etc. can be unreliable, and future work should be more cautious when basing their results on changes that developers claim to be "fixes".

ACKNOWLEDGMENTS

Most of the work by Wing Lam was done while he was interning at Microsoft. We thank August Shi, Darko Marinov, Owolabi Legunsen, and Tao Xie for their discussions about flaky tests. We also acknowledge support from Microsoft for research on flaky tests.

REFERENCES

[1] 2020. Azure Data Explorer. https://docs.microsoft.com/en-us/azure/data-explorer.
[2] 2020. Bazel. https://bazel.build.
[3] 2020. Buck. https://buckbuild.com.
[4] 2020. Data used by "A Study on the Lifecycle of Flaky Tests". https://github.com/winglam/flaky-test-lifecycle-data.
[5] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*. Hyderabad, India, 550–561.
[6] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DEFLAKER: Automatically detecting flaky tests. In *ICSE*. Gothenburg, Sweden, 433–444.
[7] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *ESEC/FSE*. Tallinn, Estonia, 830–840.
[8] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *PLDI*. Seattle, WA, USA, 153–164.
[9] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *ICSE*. Austin, TX, USA, 11–20.
[10] Martin Fowler. 2020. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html.
[11] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*. Västerås, Sweden, 1–11.
[12] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*. Florence, Italy, 55–65.
[13] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*. Baltimore, MD, USA, 223–233.
[14] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*. Madrid, Spain, 1–23.
[15] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *ICSE*. Florence, Italy, 483–493.
[16] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Roşu, and Darko Marinov. 2011. Improved multithreaded unit testing. In *ESEC/FSE*. Szeged, Hungary, 223–233.
[17] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*. Paderborn, Germany, 821–830.
[18] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*. Beijing, China, 101–111.
[19] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*. Xi'an, China, 312–322.
[20] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*. Hong Kong, 643–653.
[21] Qingzhou Luo and Grigore Roşu. 2013. EnforceMOP: A runtime property enforcement system for multithreaded programs. In *ISSTA*. Lugano, Switzerland, 156–166.
[22] Eduardo R. B. Marques, Francisco Martins, and Miguel Simões. 2014. Cooperari: A tool for cooperative testing of multithreaded Java programs. In *PPPJ*. Cracow, Poland, 200–206.
[23] John Micco. 2017. The state of continuous integration testing at Google. In *ICST*. Tokyo, Japan. https://bit.ly/2OohAip
[24] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *ESEC/FSE*. Szeged, Hungary, 496–499.
[25] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*. Shanghai, China, 1–12.
[26] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*. Lake Buena Vista, FL, USA, 857–862.
[27] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. Chicago, IL, USA, 80–90.
[28] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*. Tallinn, Estonia, 545–555.
[29] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *ICSME, NIER Track*. Madrid, Spain, 534–538.
[30] Kaiyuan Wang, Sarfraz Khurshid, and Milos Gligoric. 2018. JPR: Replaying JPF traces using standard JVM. *SIGSOFT Software Engineering Notes* 42 (2018), 1–5.
[31] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. San Jose, CA, USA, 385–396.