The Effects of Computational Resources on Flaky Tests

Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell

***Abstract*—Flaky tests are tests that non-deterministically pass and fail in unchanged code. These tests can be detrimental to developers' productivity. Particularly when tests run in continuous integration environments, the tests may be competing for access to limited computational resources (CPUs, memory etc.), and we hypothesize that resource (un)-availability may be a significant factor in the failure rate of flaky tests. We present the first assessment of the impact that computational resources have on flaky tests, including a total of 52 projects written in Java, JavaScript and Python, and 27 different resource configurations. Using a rigorous statistical methodology, we determine which tests are RAFTs (Resource-Affected Flaky Tests). We find that 46.5% of the flaky tests in our dataset are RAFTs, indicating that a substantial proportion of flaky-test failures happen depending on the resources available when running tests. We report RAFTs and configurations to avoid them to developers, and received interest to either fix the RAFTs or to improve the specifications of the projects so that tests would be run only in configurations that are unlikely to encounter RAFT failures. Although most test suites in our dataset are executed quite quickly (under one minute) in a baseline configuration, our results highlight the possibility of using this methodology to detect RAFT to reduce the cost of cloud infrastructure for reliably running larger test suites.**

I. Introduction

Flaky tests are tests that can pass and fail in repeated executions without changes to the test code or the code under test [1]. Flaky tests are detrimental to developer's productivity. In a continuous integration environment where developers run tests after making code changes, test failures signal to developers that their changes may have introduced a fault, which needs to be debugged and repaired so that all tests pass again. When a flaky test fails, the developers, unaware of the flakiness at first, may be misled to debug the test failure in the recent code changes, even though the flaky test failure is unrelated to the changes and can be due to a myriad of reasons, such as dependency on specific thread interleavings, test execution orders, etc. [1]–[3]. The negative effects of flaky tests have been reported as a substantial issue in many software companies, such as Apple [4], Ericsson [5], [6], Facebook [7], [8], Google [9]–[12], Huawei [13], Microsoft [14]–[18], and Mozilla [19], [20].

This paper makes the observation that test flakiness can often be attributed to the (un)-availability of computational resources, e.g., CPU, memory, etc. We use the term **RAFT** (Resource-Affected Flaky Test) to refer to a test that manifests flakiness under such circumstances. Intuitively, the unavailability of required but unspecified computational resources can lead to runtime errors that affect test execution. For instance, if CPU resources are unavailable, either due to test execution on a weakly equipped machine or CPU contention in a multi-processing multi-tenant setting, implicit test assumptions on the latency of asynchronous operations may be violated and lead to test failures [5], [17], [18], [21]. Overall, the (un)-availability of resources can trigger nondeterministic behavior associated with different causes of flakiness [22], e.g., Async Wait or Concurrency.

Acquiring unlimited resources is not a realistic solution to address RAFT failures as computational resources are finite, and cloud computing costs can quickly add up. Increasing resources incessantly will eventually result in diminishing returns in terms of RAFT failure prevention relative to cost. Likewise, maximizing the savings in computing resources may be disruptive and unproductive due to an increase in RAFT failures. Figure 1 illustrates the trade-off between resource availability and the likelihood of a RAFT failure. Conceptually, the "sweet-spot" region in the figure represents computational resource configurations that balance cost and failure ratio.

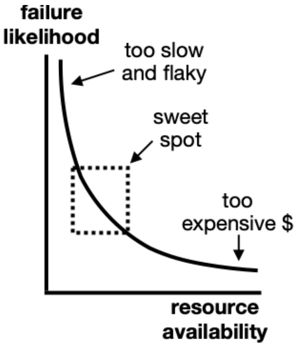


Figure 1: Trade-off between the likelihood of observing RAFT failures and resource availability.

RAFT opens an interesting perspective on the link between a controlled parameter of the execution environment—the computing resources—and the *detection* and *prevention* of flaky tests:

1) If resources are constrained, the likelihood for observing RAFT failures increases. This increase helps RAFT

Denini Silva is with Federal University of Pernambuco, Recife, PE Brazil. (e-mail: dgs@cin.ufpe.br)

Martin Gruber is with BMW Group and University of Passau, 94032 Passau, Germany. (e-mail: martin.gr.gruber@bmw.de)

Satyajit Gokhale, Ellen Arteca, Alexi Turcotte and Jonathan Bell are with Northeastern University, Boston, MA 02215 USA. (e-mails: {gokhale.sa, arteca.e, turcotte.al, jbell}@northeastern.edu)

Marcelo d'Amorim is with North Carolina State University, Raleigh, NC 27695 USA. (e-mail: mdamori@ncsu.edu)

Wing Lam is with George Mason University, Fairfax, VA 22030 USA. (e-mail: winglam@gmu.edu)

Stefan Winter is with LMU Munich and Ulm University, Germany. (e-mail: sw@stefan-winter.net)

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

detection, which is a prerequisite for localization and repair, in case the failure probability is very low in normal operation. A higher failure rate is also beneficial for debugging a flaky test.

2) If tests are known to be sensitive to resource constraints, the resource configuration of their runtime environment can be chosen in a way to reduce failure probability and thereby *prevent* RAFT from affecting developers. Future research might examine specific runtime techniques to optimize test execution.

To assess the effects of computational resources on flaky tests, we propose a rigorous statistical methodology to determine which flaky tests are resource-affected. Using this methodology, we conduct a comprehensive study involving 52 open-source projects written in Java, JavaScript and Python. We structured the study in three parts:

First, we measure the prevalence of RAFT when running tests for 300 times on 16 configurations of CPU, memory, disk, and network. *A RAFT is a test whose failure rates differ with statistical significance under throttling and non-throttling conditions (§ III-C)*. We find that nearly half of all tests found to be flaky are RAFT. These constraints intentionally impose *artificial* resource constraints that are unlikely to be observed in a real environment, but nonetheless can be used to explore the prevalence of RAFT. In a later step, we examine the prevalence of RAFT under more realistic configurations. Determining that a test is RAFT is important because it can make it easier for developers to avoid flaky failures (by providing tests with sufficient resources) and for researchers to detect flaky tests (by providing tests with fewer resources).

Second, we measure which resources have the highest impact on RAFTs. We find that CPU availability has higher importance compared to memory and much higher importance compared to disk and network. This finding is of high practical relevance for detection and prevention of RAFTs, as it is a parameter that can be controlled and scaled in real-world cloud computing configurations. In our dataset of 52 projects, we show that RAFTs are more likely to occur when the available CPU is less than 1 core and memory is less than 1GiB. This finding is relevant for tests that can run on different hardware configurations or on shared hardware with load that varies over time, which can both lead to resource contention. It also has important implications for debugging flaky test failures, as debugging is commonly conducted on a *different* machine (e.g., a developer's laptop) than the machine on which the test failed in continuous integration.

Third, we show that scaling resources beyond certain configurations yields no statistically significant improvements for preventing RAFTs. This finding has practical implications for reducing operational cost while mitigating RAFT. Furthermore, we find that RAFTs that manifest nondeterministic behavior in only one of the fifteen configurations we analyzed are rare, suggesting that a small sample of configurations can be used to detect RAFT and that increasing the number of trials should suffice to increase confidence levels that a flaky test is a RAFT. Finally, we assess the most cost-effective configurations for preventing and detecting RAFT.

Overall, we present initial, yet strong evidence of the importance of RAFTs for regression testing. As an initial study into this phenomenon, our study has limitations: most of the projects studied do not experience significant resource contention in CI, and have relatively fast-running test suites (median=52 seconds) for which acceleration or cost savings may not be meaningful. Nonetheless, this dataset supports conclusions regarding the efficacy of our methodology and the overall presence of RAFT. Our findings have several implications for developers (§ V) and researchers (§ V-C) and open an avenue for further research on flaky tests. Our dataset and scripts are publicly available under the following repository [23].

II. Background and Related Work

Flaky tests [3] have been the subject of systematic academic studies for almost a decade with numerous contributions to their detection, repair, avoidance, and tolerance at run-time. As the root causes behind the non-determinism of flaky tests are highly diverse, so are the strategies to effectively cope with them. Luo et al. [22] identified 10 diverse root causes for test flakiness across 51 affected projects from the Apache Software Foundation and derived corresponding repair strategies from fixing commits. Much of the following work to combat flaky tests consequently focused on individual root causes. iDFlakies [24] and iFixFlakies [25], for instance, have been developed as approaches for detecting and automatically repairing order-dependent flaky tests. In their work, the authors make a terminological distinction between order-dependent and non-order-dependent flaky tests, i.e., an explicit naming of the fraction their approaches aim to address vs. the totality of flaky tests. While other work is not making similarly dichotomous distinctions, the addressed root causes are commonly named explicitly, e.g., *Assumed deterministic implementations of nondeterministic specifications* [26] or *infrastructure-dependent flaky tests* [27]. A study by Ahmad et al. [28] describes practitioners' perceptions of test flakiness factors; some important factors identified related to our work include "environment understanding", "test case robustness", and "undermining network infrastructure" which can be argued to have some resource component. However, the study does not explicitly mention resource abundance or scarcity as a factor. Whereas Ahmad et al. survey developers, we expand the body of knowledge about infrastructure and flakiness by directly studying the impact of resources on test flakiness. We focus on RAFT, which provides a novel statistical methodology for examining the impact of infrastructure on test flakiness.

Besides the extensive study of RAFTs' prevalence across a large variety of popular projects along with mitigation strategies (detection *and* prevention), the focus on RAFTs conceptually distinguishes our work from a technically similar proposal by Terragni et al. [29]. While Terragni at al. also hypothesize an effect of resource unavailability on flaky test executions, their stated goal is *root-causing* in the sense of deriving a flaky test's category from a number of different possible categories, some of which are not resource-related (e.g., order-dependency). Our work, in contrast, is focusing on

```
1  @Test
2  public void testIssue() throws Exception {
3    server.start();
4    countServerDownLatch.await();
5    webSocket0.connectBlocking();
6    assertTrue("webSocket.isOpen()",webSocket0.isOpen());
7    webSocket0.close();
8    assertTrue("webSocket.isClosing()",webSocket0.isClosing()
       );
9    countDownLatch0.await();
10   assertTrue("webSocket.isClosed()",webSocket0.isClosed());
11   webSocket1.connectBlocking();
12   assertTrue("webSocket.isOpen()",webSocket1.isOpen());
13   webSocket1.closeConnection(CloseFrame.ABNORMAL_CLOSE, "
       Abnormal close!");
14   assertTrue("webSocket.isClosed()",webSocket1.isClosed());
15   server.stop();
16 }
```

Figure 2: Example RAFT from the `Issue677Test` class in the `Java-WebSocket` project [30].

resource effects and explores two mitigation strategies. For this reason, our work leverages resource control from Linux control groups, which provides uniform resource access control over the entire duration of a test execution, whereas other work (e.g., Terrani et al.'s [29] and Shaker [21]) rely on dynamic load generation, with which tests compete for resources.

A. Example RAFT

Figure 2 shows an example of a RAFT. The test is from the open-source project `Java-WebSocket` that provides an implementation of the asynchronous websocket protocol in Java. The test, `Issue677Test` checks for a specific regression, where the method `isClosed` continues to return the intermediate state `Closing` even after the socket is disconnected, and the state should be `Closed`. Several lines not shown in the figure create the objects `server`, `webSocket0`, `countServerDownLatch` and `countDownLatch0`. In this test, Line 3 starts the server, calling the asynchronous `start` method. The test fixture instruments the server to notify the `countServerDownLatch` once the server has completed starting, so that the test waits at Line 4 until the server is ready to proceed, avoiding a potential flaky failure from the test connecting to the server before the server is started. Then, the websocket client is closed (Line 7) and the test asserts that the client state is set to `isClosing` (Line 8). Since `close` is an asynchronous method, the test instruments the client code to notify the test via the `countDownLatch0` to track when the socket is actually closed, such that the assertion on Line 10 does not race with the close operation (again, avoiding flakiness). Despite these efforts at synchronization to avoid flakiness, this test is nonetheless flaky, as the assertion on `webSocket.isClosing()` (Line 8) *also* can race with the `close` operation. Rather than set the state to *isClosing* immediately, the *close* method does so in another thread, without synchronization.

When this test "gets lucky," the `close` method gets to set `isClosing` before the assertion tests it. However, if there are insufficient CPU resources to execute the code in that other thread quickly enough, the assertion runs before `isClosing` is set. When we run this test on a 4 CPU and 16GiB RAM virtual machine, we find that this test fails 14 out of 300 times. However, when run on a virtual machine with access to only 1/10 of a CPU, it fails 54 out of 300 times, a significant difference ($p = 0.000198$). We submitted a pull request to the owner of the project explaining why that specific assertion was unreliable and recommended its removal (the test has other assertions). The owner accepted the PR.

III. Methods and Objects of Analysis

This section describes the projects we used (§ III-A), the setup of our experiments (§ III-B), and the research questions we posed (§ III-C).

A. Projects

Our empirical study includes projects written in three languages: Java, Python and JavaScript. For each language, we selected projects by examining the literature to identify projects previously studied in the context of flakiness. We included projects studied by prior work if we could build the project, run the tests, and parse the test output. In cases where projects had missing dependencies (or other infrastructure-related failures), we spent up to three hours per-project manually debugging them, improving our tools if necessary. For each project in our dataset, we create Docker containers that have all of the project's dependencies included, ensuring durable reproducibility and reducing the effort needed by researchers in the future to build on our results.

1) Java: The corpus of Java projects consists of 30 GitHub projects selected from two datasets: 15 projects from the FlakeFlagger dataset [31] and 15 projects from the Lam et al. dataset [32]. We use the same versions of each of these projects as studied in our and others' prior work. This set of projects has been extensively studied in the context of flaky tests, and was originally built by searching GitHub for issues or commits related to flaky tests. We excluded four projects from the original FlakeFlagger dataset (three of which manifested deadlocks and one had a broken build) and excluded five projects duplicated in the Lam et al. dataset (all of which are also included in the FlakeFlagger dataset).

2) Python: The corpus of Python projects consists of 12 projects selected from Parry et al.'s recent studies [33], [34]. These Python projects were selected at random from a list of projects critical to open-source infrastructure. We first attempted to use the exact same versions of these projects that had been studied in prior work, but despite significant and generous assistance from the authors, were unable to successfully build those old versions due to missing dependencies. We *did* succeed at building the most recent revision of 21 of these projects (excluding five), and created container images with those dependencies cached to ensure durable reproducibility. For nine of these projects, we did not observe a single flaky test during any test execution, leaving us with 12 Python projects, that we were able to build and for which we observed at least one flaky test.

3) JavaScript: We used a similar methodology to select 10 JavaScript (JS) projects, beginning by examining the projects studied in Barbosa et al.'s investigation of flaky tests across programming languages (six JS projects with at least five flaky test) [35], and Yost's flaky test detection work [36] (58 JS projects). We used our NPM-Filter infrastructure [37] fort building the projects, running their test suites and parsing the test results. We ran each project under NPM-Filter, and included in our corpus each project that completed within three hours, and for which NPM-Filter could parse test results (i.e., those using the Mocha or Jest test runners). We supplemented this set of JavaScript projects with three projects that we had previously encountered flaky tests in: `ngrok`, `IcedFrisby` and `twilio-video-app-react`. Ultimately, this resulted in a corpus of 10 JavaScript projects with flaky tests.

These datasets are a part of recent research on flaky test detection, which makes them ideal targets for our study and provides baselines against which our results can be compared. Figure 3 shows the distribution of flake rates for the flaky tests in these projects. We note that these projects do *not* provide guidelines for the resources needed to run tests. Hence, we calculate these statistics by executing each test suite 300 times in a resource configuration comparable to the default configuration provided by CI platforms (2 CPU cores and 8GB RAM). The flake rate is the number of failing runs of a test divided by the total number of trials. Overall, we observe that our dataset contains some tests that fail very frequently in this configuration (in more than half of the runs), while others fail more infrequently (in fewer than a quarter of the runs). Table III lists all of these projects in alphabetical order grouped by language. Our supplementary artifact [23] contains URLs for the projects analyzed including corresponding revisions used, along with links to docker images that contain the projects packaged with all necessary dependencies to reproduce their test suites. Table IV shows the average test suite execution time for each of the projects in our dataset, ranging from some as fast as just under a second, to others as slow as 24 minutes. The median test suite execution time in our dataset is 52 seconds.

B. Experimental setup

The experiment consists of two phases:

Phase I: This phase is designed to identify the most prominent resource(s) responsible for RAFT.

Phase II: This phase is designed to identify the most economically prudent real-world configurations for detection and prevention of RAFT.

During both phases, the base machine consists of a virtual machine in our VMWare private cloud. All experiments are run on virtual machines that are allocated 4 CPU cores and 16 GiB of RAM. Within these virtual machines, we run the test suites in Docker containers, using Docker to further restrict the resources available to the test suite. Each experiment is implemented as a series of "jobs," where each job includes the execution of one test suite under one resource-availability configuration to isolate the effects of resource availability on individual test suite executions. We control the test order in test suite executions to avoid effects from order dependencies. The invocation of the experiments is managed using Slurm [38], which schedules the execution of each job of each experiment on our cluster. To prevent interference between experiments, only a single container was run at a time within any virtual machine. It is worth noting that flaky test failures are inherently nondeterministic. This randomness can be problematic due to its potential to skew results on a particularly lucky (or unlucky) run. For that reason, we run each of the 52 projects on every configuration 300 times and record the test failures for each run.

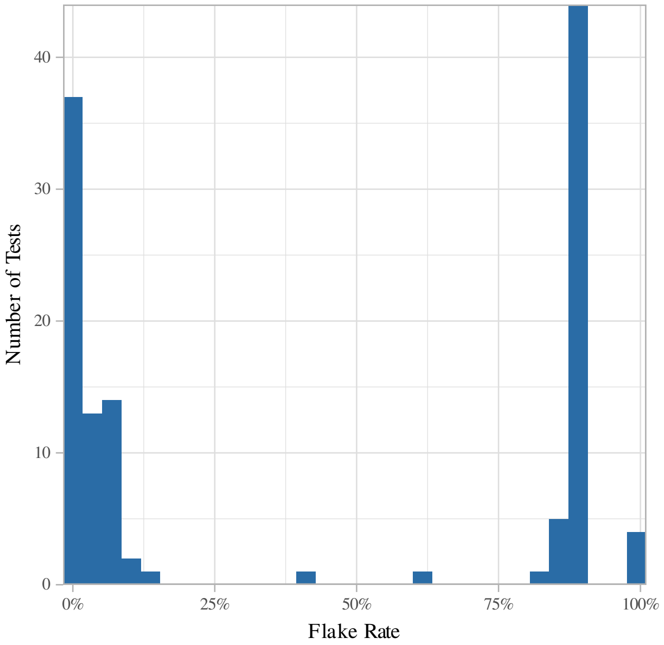


Figure 3: Flake rates for the flaky tests included in this study. Each flaky test was run 300 times in a resource configuration comparable to the default provided by continuous integration platforms (2 CPU cores, 8 GB RAM). The flake rate is computed as the ratio of failing runs to total runs.

Table I: Throttling configurations for Phase I. The highlighted row shows the default configuration (no throttling). Empty cells indicate that the value of the corresponding cell is equivalent to that of the default configuration (Baseline).

#	C	M	D	N
Baseline	4	16	Unrestricted	Unrestricted
(C)	0.1			
(M)		0.5		
(D)			50/100 Kbps	
(N)				1500/512 Kbps
(CM)	0.1	0.5		
(CN)	0.1			1500/512 Kbps
(MN)		0.5		1500/512 Kbps
(CD)	0.1		50/100 Kbps	
(MD)		0.5	50/100 Kbps	
(DN)			50/100 Kbps	1500/512 Kbps
(CMN)	0.1	0.5		1500/512 Kbps
(CMD)	0.1	0.5	50/100 Kbps	
(CDN)	0.1		50/100 Kbps	1500/512 Kbps
(MDN)		0.5	50/100 Kbps	1500/512 Kbps
(CMDN)	0.1	0.5	50/100 Kbps	1500/512 Kbps

Phase I: To understand the impact of different resources (CPU, Memory, Disk, and Network), it is necessary to have control over them and have the ability to restrict them independently. Table I shows the complete list of throttling configurations used during Phase I of our study. Column "#" shows the configuration ID, column "C" shows the number of CPUs, and column "M" shows the amount of memory in GiBs. The column "D" (abbreviates Disk) shows the limited rate of IO operations per second and the throughput in kilobit per second (Kbps), respectively. Finally, the column "N" (abbreviates Network) shows the network limit for download and upload in Kbps respectively. The options for CPU, RAM, and disk throttling are set using the Docker CLI. The option for networking throttling is set using Wondershaper.[1]

The first row on Table I shows the default configuration, where resources are *not* throttled. In our baseline configuration, 4 CPU cores and 16 GiB of main memory are provided. For comparison, the entry-level machines that GitHub Actions provide include 2 CPUs and 7 GiB of memory, whereas Google Cloud Build machines can be configured with as little as 1 CPU and 4GiB of memory. Section V-D elaborates on common entry-level configurations from other cloud vendors: developers may turn away from the default cloud CI configuration to meet higher resource requirements or optimize cost. The non-default configurations (C)-(CMDN) from Table I modify the values assigned to one or more resources. We chose very small values to assign to each configuration option (i.e., resource) with the goal of running the tests under "limit" conditions. The precise values have been obtained by gradually increasing from extremely low values until the projects in our study were able to run (i.e., no failures for builds or a majority of tests). With 4 resources that can be restricted and a single limit value the resource is restricted to, there are 2^4 possible combinations of restrictions, which correspond to the 16 rows in Table I. The configurations (C)-(N) throttle only a single resource at a time, and are useful to understand the impact of individual resources on flaky failures. We also consider all combinations of those parameters, which helps us to answer whether certain tests are RAFTs only when multiple resources are constrained simultaneously. These are *extreme* configurations where we reduce resource availability to an extent that is likely greater than would be experienced in day-to-day development. These extreme configurations allow us to provide something of a bound on the number of RAFT, as we expect that some tests will be more likely to present as flaky under greater restrictions. In order to examine resource configurations that developers are more likely to experience on a day-to-day basis, we designed Phase II of our study.

Phase II: In order to identify the most cost effective real-world cloud computing configurations for detection and prevention of RAFTs, we examine resource configurations that more closely match those available by major cloud providers. Cloud providers offer flexible pricing for on-demand containers-as-a-service, e.g. AWS Fargate [39], Google Kubernetes Engine [40], and Azure Kubernetes Service [41]. These services are priced by CPU and memory specifications, and provide "standard" disk and network access services. During Phase I, we conclude that CPU has a greater influence on test flakiness compared to memory, disk and network. As a result, in Phase II, we consider configurations with a different number of CPUs, assigning the lowest and the highest memory options available on AWS for each configuration. No restrictions are imposed on the disk and network during this phase. As in Phase I, we ran each test suite 300 times. Table II shows the complete list of configurations used during Phase II of our study. Column "#" shows the configuration id, column "CPU" shows the number of CPUs, column "Mem(GiB)" shows the amount of memory in GiBs. Column "Cost ($/hr)" shows the AWS computing cost per hour of a given CPU and Memory configuration on AWS Fargate [39] serverless compute engine. In AWS Fargate, developers submit Dockers images to the engine and pay for compute resources when used. We consider 12 combinations of CPU, ranging from 0.1 to 4, and of memory, ranging from 1 GiB to 16 GiB. The AWS computing cost (measured in USD per hour) varies with the quality of the service and the configuration requested [42]. For a given configuration, the cost of the "spot" service is lower compared to the cost of the "on-demand" service. Whereas the "on-demand" service provides guaranteed availability of the container, a "spot" container may be interrupted and canceled by the service provider to shed their load during peak usage times. However, the significant savings may make it attractive for running test suites in CI, where a canceled test suite can be restarted on another container.

Table II: AWS configurations sorted by cost [42]. Disk and Network are unrestricted.

#	CPU	Mem (GiB)	Cost (USD/hour)	
			spot	on-demand
1	0.1	1	0.002548	0.008493
2	0.1	2	0.003881	0.012938
3	0.25	2	0.005703	0.019010
4	0.5	2	0.008739	0.029130
5	0.5	4	0.011406	0.038020
6	1	4	0.017478	0.058260
7	1	8	0.022812	0.076040
8	2	4	0.029622	0.098740
9	2	8	0.034956	0.116520
10	2	16	0.045624	0.152080
11	4	8	0.059244	0.197480
12	4	16	0.069912	0.233040

C. Research questions and methodology

We aim to answer the following key research questions:

RQ1. *How prevalent are RAFTs?*

Rationale. This question is important to justify further investigation on *Resource-Affected Flaky Tests* (RAFTs). If we find that none of the flakiness can be attributed to resource starvation, then further investigation is meaningless. To understand the prevalence of RAFTs in flaky test failures, we aim to answer two key questions:

RQ1.1. How many of the flaky test failures can be attributed to resource starvation?

[1] https://github.com/magnific0/wondershaper

RQ1.2. How sensitive are RAFT failures to resource starvation?

RQ1.1 aims to distinguish RAFT failures from other kinds of failure to establish their prevalence. RQ1.2 aims to quantify the effect of resource starvation on RAFT failures to establish how likely these failures are under resource throttling.

Methodology. To answer RQ1, we consider the data for the following attributes for each project: (i) the number of flaky tests identified under full resource availability, (ii) the number of flaky tests identified under all test execution configurations combined (Table I), and (iii) the number of test failures which can be considered RAFTs. Since a definition for RAFTs does not exist, we present the first quantifiable definition of RAFTs.

Definition: A *Resource-Affected Flaky Test* (RAFT) is a flaky test that has a statistically different failure rate when resources are constrained compared to an unconstrained test execution. We use Pearson's chi-squared test to determine whether the failure rate is statistically different, accepting that difference as significant only at a level of $p < 0.05$. To reduce the false discovery rate, we use the Benjamini-Hochberg procedure to adjust p-values. For a given throttling configuration, a test can be a RAFT only if it passed at least once under that same configuration.

We use the three attributes above to answer RQ1.1. To answer RQ1.2, we consider the increase in failure rates for every unique test case. Such increase is defined as the ratio $f_i/max(f_1, 1)$, where f_i is the number of failures under the most failure-inducing throttling configuration and f_1 is the number of failures under the configuration with no resource throttling. The ratios are then grouped by different levels of increase in failure rate.

RQ2. *Which resources have the strongest influence on flakiness?*

Rationale. This question is important to justify further investigation of the relationship between resource availability and test flakiness. If we find that the relationship is weak, then further investigation is meaningless. To understand the effect of machine resources on flaky test failures, we aim to answer two questions:

RQ2.1. What resources are most common at triggering flaky test failures?

RQ2.2. Are some flaky tests only detected when using different combinations of resources?

RQ2.1 aims to study the effect that throttling individual resources has on flaky test failures. RQ2.2 aims to study the effect that throttling combinations of resources has on flaky test failures and whether this produces results that are significantly different to throttling individual resources. It is important to explore each resource independently and in combinations to understand their impact. The resources or combinations with the most impact can then be chosen for further analysis.

Methodology. To answer RQ2, we compare the number of RAFTs detected under each throttling configuration shown in Table I. Each throttling configuration limits availability of one to four resources. We compare the number of RAFTs detected by each configuration, analyzing the configurations that detect each RAFT.

RQ3. *Which configuration best saves money while running the test suite <u>to prevent RAFTs</u>?*

Rationale. In a typical usage of continuous integration, developers want to simultaneously (1) avoid flaky tests to reliably determine whether a bug is present in code when observing test failures and to (2) run tests efficiently, i.e., maximize test runs per amount of money. This question focuses on this scenario. More precisely, it investigates which resource configurations give the lowest flaky test disruption per amount of money for given a project.

Methodology. To answer RQ1 and RQ2, we examined the total RAFTs detected across all 300 test suite invocations. To answer RQ3, we study instead the number of test suite invocations (i.e., builds) that have at least one flaky-test failure. We consider two metrics for every resource configuration listed in Table II: (i) the number of builds with test failures across all test runs (as a proxy for reliability) and (ii) the price per-test suite run. We calculate the price per-test suite run by multiplying the average time to run the project's test suite (as reported by the build system) by the "on-demand" AWS Fargate cost shown in Table II. The configurations with the least number of build failures are considered the most reliable. However, there may be other configurations which have slightly higher rate of build failures but are more cost effective. It is worth noting that configurations with a lower hourly rate can take longer to complete due to limited resources, resulting in a higher cost for each build compared to an expensive but fast configuration. For every configuration in Table II, we consider the number of projects for which it had the best price, best reliability, or both.

RQ4. *Which configuration best saves money while running the test suite <u>to detect RAFTs</u>?*

Rationale. In another use case, developers may want to (1) detect flaky tests in advance (i.e., before observing potentially spurious failures during regression runs) and (2) run tests efficiently. This question focuses on this scenario. More precisely, it investigates which configurations maximize the ability of test runs to detect flaky tests while keeping costs at a minimum.

Methodology. To answer RQ4, we consider two metrics for every resource configuration listed in Table II: (i) the number of flaky test failures (as a proxy for reliability of detection) and (ii) the price per run. As with RQ3, we calculate the price per-run by multiplying the average time to run the project's test suite (as reported by the build system) by the "on-demand" AWS Fargate cost shown in Table II. Intuitively, cheaper configurations are more likely to detect flaky test failures, but they may not be the most cost effective due to slower execution times. Furthermore, some configurations may be entirely unusable for some projects - for example, when a project's tests require some minimum amount of memory to run at all. Hence, it is necessary to consider computing cost for this analysis. For every configuration in II, we consider the number of projects for which it had the best price, best detection, or both.

Table III: For each project with flaky tests, we report the baseline number of flaky tests identified without resource throttling (Baseline Flaky) and the number of flaky tests identified as RAFT under each throttling condition. Throttling conditions are identified by the resources throttled, i.e., CPU(C), Memory(M), Disk(D), Network(N), and combinations thereof. "Total, All Runs" summarises all flaky tests (and RAFT) detected *including* the Phase II (AWS) configurations. Some projects resulted in catastrophic failures under certain configurations that are indicated as ”-”.

		Baseline	Flaky Tests Identified as RAFT Under Throttling Conditions															Total, All Runs	
Language	Project	Flaky	(C)	(M)	(D)	(N)	(CM)	(CN)	(MN)	(CD)	(MD)	(DN)	(CMN)	(CMD)	(CDN)	(MDN)	(CMDN)	Flaky	RAFTs
Java	assertj-core	1	**1**	-	0	0	**1**	**1**	0	**1**	-	0	-	**1**	**1**	0	**1**	3	1
	carbon-apimgt	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	1	1
	commons-exec	0	**1**	0	0	0	**1**	**1**	0	**1**	0	0	**1**	**1**	**1**	0	**1**	3	1
	db-scheduler	5	2	0	0	0	2	**6**	0	2	0	0	**6**	2	2	0	**6**	7	6
	delight-nashorn-sandbox	1	17	1	0	0	**24**	20	0	16	0	0	**24**	23	19	0	23	27	24
	elastic-job-lite	0	0	-	0	0	-	0	0	0	0	**1**	-	-	0	-	-	1	1
	esper	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	excelastic	1	**1**	0	0	0	**1**	**1**	0	**1**	0	0	**1**	**1**	**1**	0	**1**	2	1
	fastjson	0	0	-	0	0	**2**	0	-	0	-	0	**2**	**2**	1	-	**2**	3	2
	fluent-logger-java	0	1	0	0	0	**2**	1	0	1	0	0	**2**	**2**	1	0	**2**	5	2
	handlebars.java	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	hector	2	0	0	0	0	0	-	0	-	0	0	-	0	-	0	-	2	0
	http-request	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0
	httpcore	4	1	0	0	0	1	1	0	1	0	0	**2**	1	1	0	1	22	2
	hutool	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
	incubator-dubbo	3	23	-	0	0	-	**25**	-	**25**	-	0	-	-	22	-	-	55	27
	java-websocket	22	16	7	0	1	22	20	5	18	8	0	**24**	**24**	19	8	22	37	32
	logback	6	7	0	0	0	7	6	0	8	0	0	**9**	-	6	0	8	28	12
	luwak	0	0	0	0	0	**1**	**1**	0	**1**	0	0	0	0	**1**	0	0	4	2
	ninja	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0
	noxy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	orbit	2	**2**	0	0	0	**2**	**2**	0	**2**	0	0	**2**	**2**	**2**	0	**2**	6	2
	oryx	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
	riptide	0	1	**2**	0	0	1	1	0	1	0	0	1	1	1	0	1	5	3
	rxjava2-extras	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	1
	spring-boot	0	0	-	0	0	-	0	-	0	-	0	-	-	0	-	-	3	0
	timely	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
	wro4j	7	**5**	-	0	0	-	**5**	-	**5**	-	0	-	-	**5**	-	-	13	6
	yawp	1	**1**	0	0	0	**1**	**1**	0	**1**	0	0	**1**	**1**	**1**	0	**1**	1	1
	zxing	2	0	0	0	0	0	0	0	0	0	0	-	0	0	0	0	2	0
	30 Projects Total	70	79	10	0	1	68	**93**	5	84	8	1	75	61	84	8	71	256	128
JavaScript	apollo-client-devtools	0	11	-	0	0	-	8	-	10	-	0	-	-	**13**	-	-	21	13
	IcedFrisby	1	0	0	0	0	0	9	**10**	9	**10**	9	**10**	0	0	**10**	9	16	11
	javascript-action	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
	ngrok	2	0	0	0	0	0	0	0	1	0	0	0	1	**2**	0	1	8	2
	preset-modules	0	**2**	0	0	0	1	1	0	**2**	0	0	1	1	1	0	1	2	2
	react-datetime	0	**1**	0	0	0	**1**	**1**	0	**1**	0	0	**1**	**1**	**1**	0	**1**	2	2
	react-native	1	3	0	0	0	3	3	0	3	0	0	-	3	**4**	0	3	17	10
	shields	0	**5**	0	0	0	**5**	**5**	0	**5**	0	0	**5**	**5**	**5**	0	**5**	6	6
	tippyjs-react	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	1	1
	twilio-video-app-react	0	6	-	0	0	-	6	-	**8**	-	0	-	-	**8**	-	-	18	8
	10 Projects Total	4	28	0	0	0	10	33	10	**39**	10	9	17	12	35	10	20	92	56
Python	celery	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	conan	0	0	0	**2**	0	0	0	0	**2**	**2**	**2**	0	**2**	**2**	**2**	**2**	6	2
	electrum	0	**1**	0	0	0	**1**	0	0	**1**	0	0	**1**	**1**	**1**	0	**1**	1	1
	fonttools	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	ipython	0	0	0	0	0	0	0	0	**3**	0	0	0	**3**	**3**	0	**3**	4	3
	loguru	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42	0
	mitmproxy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	requests	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
	seaborn	0	0	**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	**1**	0	**1**	**1**	2	1
	setuptools	8	0	52	10	0	52	0	52	13	60	11	52	56	14	**62**	57	177	89
	sunpy	0	0	**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	**1**	0	**1**	**1**	11	1
	xonsh	0	**1**	0	0	0	**1**	**1**	0	**1**	**1**	**1**	0	0	0	**1**	0	10	2
	12 Projects Total	12	2	54	12	0	56	1	54	20	65	14	55	64	20	**67**	65	260	99
52 Projects Total		86	109	64	12	1	134	127	69	143	83	24	147	137	139	85	**156**	608	283

IV. Results

A. Answering RQ1: How prevalent are RAFTs?

This research question evaluates prevalence of Resource-Affected Flaky Tests (RAFTs) among flaky tests. Table III summarizes the results of 300 test runs on each of the 52 projects for every throttling configuration in Table I. For every project in the table, the column "Baseline Flaky" contains the number of flaky tests identified under no resource throttling. The columns "Flaky" and "RAFTs" under "Total, All Runs" represent the total number of flaky tests identified across all configurations, and those unique tests which can be considered RAFTs, respectively. The remaining columns show how many RAFTs were observed by throttling different kinds of resources: CPU (C), Memory (M), Disk (D), or Network (N), and combinations thereof.

1) ***RQ1.1** How many of the flaky test failures can be attributed to resource starvation?:* With no resource throttling, we observed a total of 86 flaky-test failures when running the tests of each project for 300 times and aggregating results across all 52 projects. Across all configurations, we observed a total of 608 flaky test failures, of which 283 tests were classified as RAFTs. The highest number of RAFTs identified in a single Java project is 32 in `java-websocket` (=86.48% of the total of flaky tests on that project). Within the JavaScript projects `apollo-client-devtools` showed the highest number of RAFTs (13, or 61.90% of total flaky tests in that project), and within the Python projects, `setuptools` showed the highest number of RAFTs (89, or 50.28% of the total flaky tests in that project). We observed no RAFTs in 15 projects of 52 projects. Note that the number of test runs for all combinations of resource throttling is significantly greater than that for no resource throttling (4,500=300*15 versus 300). We run baseline configuration and every other configuration for 300 times. This, in conjunction with potentially increased failures in RAFTs due to throttling accounts for the difference in the numbers between the columns "Baseline Flaky" and "Total, All Runs/Flaky".

> *Summary:* Of all flaky tests detected in our study, we find that 46.5%(=283/608) of them are RAFTs.

2) ***RQ1.2** How sensitive are RAFT failures to resource starvation?:* The barplot in Figure 4 shows the distribution of tests on various "resource-affectedness" levels for each project that contains RAFTs. A level is defined as the increase in failure rate relative to the baseline "no throttling" configuration. The colors indicate different levels. For example, dark green denotes a test that is not resource affected, while red indicates a test that is severely resource affected. The length of each bar denotes the number of flaky tests found with throttling runs, i.e., it corresponds to the value in column "Total, All Runs/Flaky" on Table III.

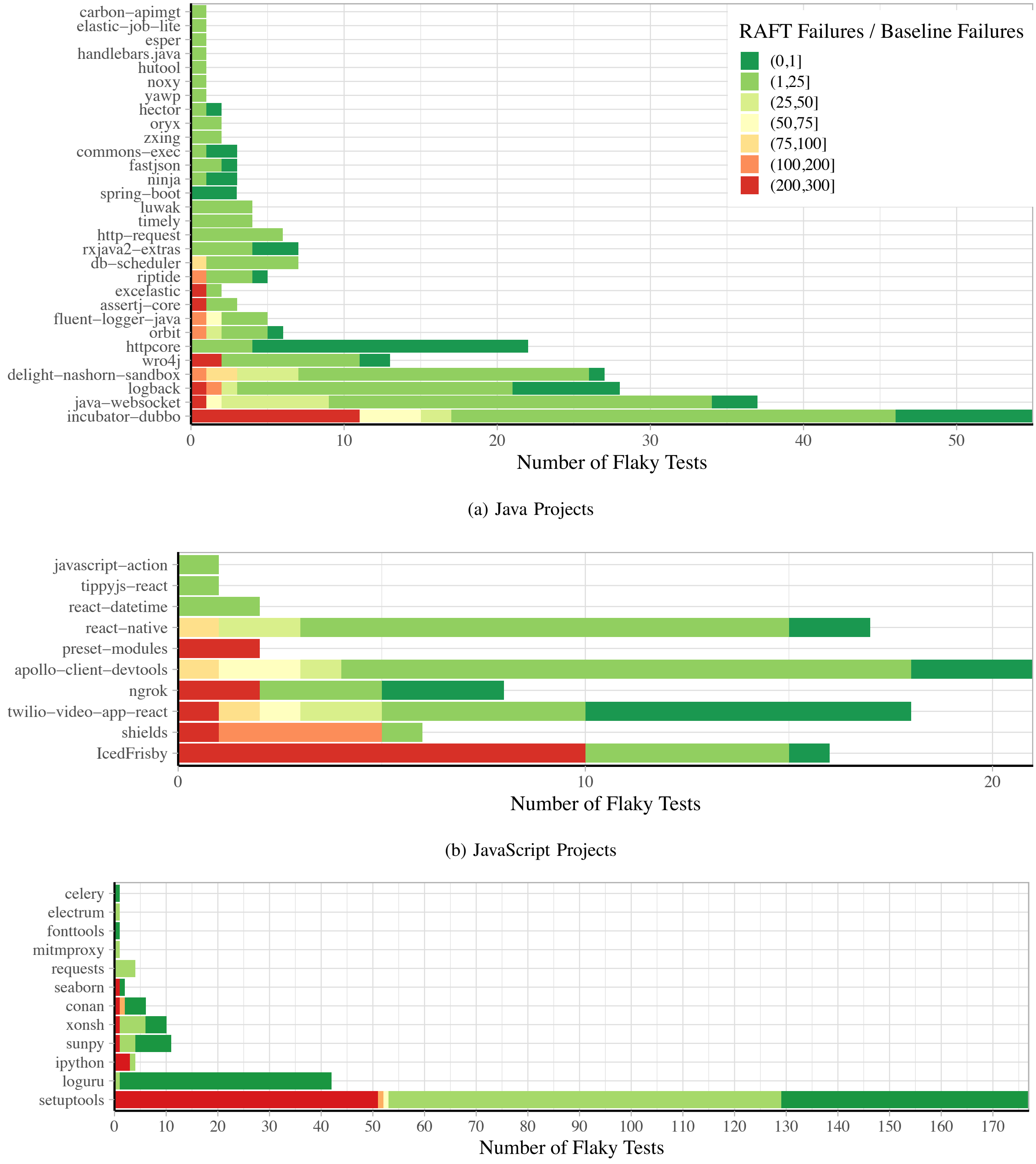
Based on Figure 4, we observe that the level of resource-affectedness varies with each project. Most projects contain RAFTs which are slightly affected by resources with an increase in failure rate of less than 25x (shown in light green). Some projects such as `incubator-dubbo` for Java, `IcedFrisby` for JavaScript, and `setupTools` for Python have a large number of RAFTs that are heavily affected by resource availability with an increase in failure rate of over 200x. RAFTs that are heavily affected by resources are quite uncommon and mostly exist in small numbers in few projects. We should note that RAFTs that are heavily affected by resources may already be well-known to developers, as they are clearly very sensitive to resource availability, and are quite likely to fail if resources are unavailable. On the contrary, the most commonly-occurring RAFTs in our experiment (shown in light green, those that increased in failure rates more marginally), may be the most dangerous, since developers are less likely to make the connection between the flaky-test failures and resource availability. Heavily affected RAFTs on the other hand, can fail two or hundreds of times more frequently than normal and are therefore easier to identify. Section V-A describes developer feedback to the RAFTs that we identify.

> *Summary:* Most commonly, RAFTs are slightly affected by resources. In a 0-300 scale of resource-affectedness, the most predominant range is 1-50.

B. Answering RQ2: Which resources have the strongest influence on flakiness?

This research question evaluates the impact of individual resources and their combination on test flakiness. Table III summarizes the test failures for 300 runs on all configurations. The columns (C), (M), (D), and (N) contains the number of test failures for throttling of individual resources. The columns after those show the test failures for combinations of these resources.

1) ***RQ2.1** What resources are most common at triggering flaky test failures?:* We begin by examining the characteristics by language. Across all 30 Java projects, we observed 82 test failures under CPU throttling, 10 under memory throttling, one under disk throttling, and one under network throttling. Across all 10 JavaScript projects, we see a similar pattern: we observed 28 failures under CPU throttling, zero under memory throttling, 1 under disk throttling, and 1 under network throttling. This trend breaks among the 12 Python projects, apparently due to the influence of a single project (setuptools): we observe only two failures under CPU throttling, 54 under memory throttling, 13 under disk throttling, and zero under network throttling. This behavior might be explained by different minimum memory requirements across projects: our "memory" throttling configuration allows only 512MB of RAM, which was insufficient to even run tests for some of the projects with the most RAFTs in other languages (e.g. incubator-dubbo, apollo-client-dev-tools), whereas this amount appeared to be just enough to run the tests in setuptools, albeit sufficiently little to cause many flaky test failures. However, what is clear from the results is that disk and network throttling play a much more minor role in causing



(a) Java Projects

(b) JavaScript Projects

(c) Python Projects

Figure 4: Just how resource affected are these flaky tests? For each project with flaky tests, we show the failure increase rate from the baseline “no throttling” configuration to the most failure-inducing resource throttling condition.

flaky test failures. Hence, we conclude that CPU starvation is the most significant and ubiquitous factor for increasing test failures, while memory throttling may also be impactful.

> *Summary:* The resource that triggers flakiness most frequently in Java and JavaScript projects in our dataset is CPU, and in Python is memory.

2) ***RQ2.2*** *Are some flaky tests only detected when using different combinations of resources?:* Recall that, according to our definition, a RAFT is a test that has a statistically greater failure rate in at least one resource-throttled configuration, as compared to its baseline failure rate (under no throttling). Of the total of 283, we identified 24 tests that were RAFTs in only one configuration. In each case, the absolute difference in failures was relatively small, ranging from an increase between seven and 23 additional failures observed under the single configuration that exposed the test as RAFT and the baseline failure count. To further investigate these tests, we use Pearson's chi-squared test to determine whether there were other throttling configurations that resulted in failure rates that were statistically indistinguishable from that single RAFT case. In all but three cases, we found at least one other throttling configuration that induced the RAFT to fail at a rate that was indistinguishable from both the RAFT-inducing configuration and the baseline configuration. Two of these tests belonged to the Python project `setuptools`, and were RAFTs only in the CMD configuration. The last test belonged to the Java project `riptide`, and was flaky only in memory-throttling configurations, failing persistently in CPU-throttled configurations. We conclude that it is unlikely that it is necessary to examine every resource configuration in order to detect RAFTs, and simply increasing the number of trials may be sufficient to increase confidence levels. Nonetheless, in projects where tests are known to rely on disk input/output (such as in the case of the `setuptools` project), adding disk throttling combinations may help to detect RAFTs.

> *Summary:* RAFTs rarely manifest only in specific throttling configurations. Of the 283 RAFTs, 24 of them manifested only in one of the 15 configurations.

C. Answering RQ3: Which configuration best saves money while running the test suite to prevent RAFTs?

This research question evaluates the reliability of test execution and cost for each AWS configuration. To answer this question, we analyzed the percentage of build failures and the price to run every project on each AWS configuration in Table II. Figure 5 summarizes the results as a stacked bar chart ranking each configuration on price, reliability, and both. Some projects may run with equal reliability on more than one configuration but have different price points. In such cases, the project is only shown under the "Best Reliability and Price" category. Conversely, some projects may show significantly different behavior at drastically different price points on different configurations.

Figure 5 shows that the best configuration for reliability and price largely depends on individual projects. We observe that the price of builds does not scale linearly with the price of configurations. Configurations with lower resources generally have a lower billing rate. However, the constrained resources can make individual builds significantly slower. As a result, it can often be more expensive to run builds on cheaper configurations compared to those with slightly higher billing rates. Whether or not this is the case depends on how a project utilizes resources. If a project is CPU-heavy and the CPU is constrained, this may result in longer execution times and higher costs. If, on the other hand, a project is I/O-heavy, limiting CPU time does not affect execution time and costs as much. In addition, some projects could not be run on low-resource configurations due to catastrophic failures (e.g., deterministically running out of memory). The minimal amount of resources required for execution are also project-specific. Hence, the most reliable and most cost-effective configurations vary based on the projects. We observed that the configuration "CPU 0.5 and RAM 2GiB" is the most cost effective configuration, followed by "CPU 2 and RAM 4GiB" and "CPU 1 and RAM 4GiB." We observed that the most reliable configurations are "CPU 0.5 and RAM 2GiB" and "CPU 2 and RAM 4GiB". A table showing the price and reliability of each configuration for each project is included in the appendix to this article.

Adopting a lower-resource configuration can also lead to an increase in latency, as tests may run slower. Table IV shows the average test suite execution time for each of the projects, along with the slowdown under each configuration, with the slowdown of the cheapest, most reliable configuration bolded. Not surprisingly, reducing the number of CPU cores available has a significant impact on feedback time. We discuss tradeoffs in selecting CI resource configurations further in Section V-D.

> *Summary:* The most cost-effective configuration to prevent RAFTs largely depends on the project.

D. Answering RQ4: Which configuration best saves money while running the test suite to detect RAFTs?

This research question evaluates the reliability of test failures and cost for each AWS configuration. To answer this question, we analyzed the number of test failures and the cost to run every project on each AWS configuration in Table II. We expect that the total cost savings will be quite small for the projects that we studied, as Table IV shows that the median test suite execution time was just 52 seconds (Section V-D1 discusses the absolute cost savings and estimates a maximum of $\$0.48/month$). Figure 6 summarizes the results as a stacked bar chart ranking each configuration on price, detection, and both. As discussed in the prior sub-section, the cost for individual runs depends on their execution time and can, thus, vary significantly for different configurations. The cost

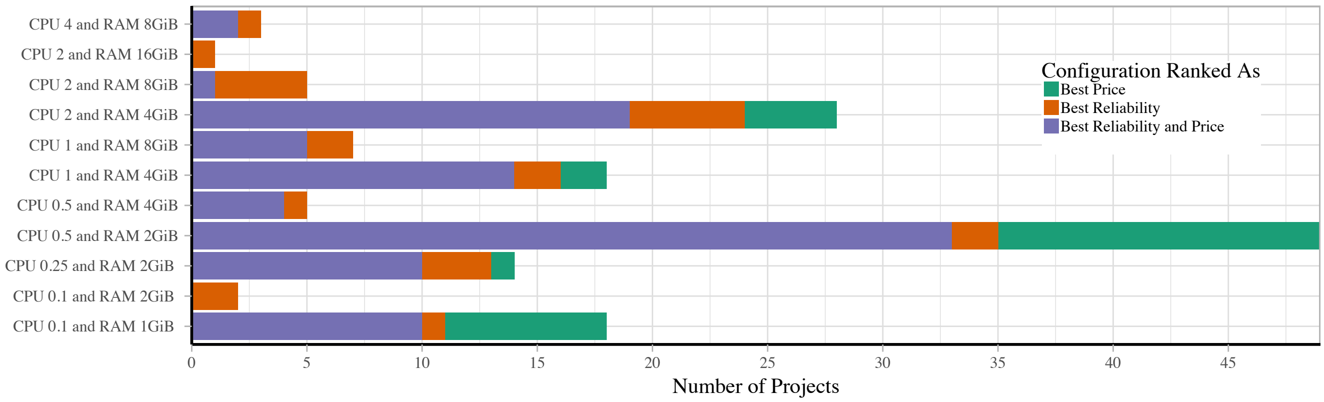


Figure 5: What are the best resource configurations to prevent flaky failures? For each configuration that we analyzed, we show the number of times that it was the best at avoiding flaky failures, the best in terms of price, or the best in terms of both. If a configuration was tied for best in terms of reliability for a project, we select the cheaper one. We hide configurations that were not optimal on either dimension.

of execution on cheaper configurations can be more than the cost of execution on expensive configurations because weaker configurations entail a longer execution time. Similar to RQ3, Figure 6 shows that the best configuration for detecting flaky test failures and obtaining best price depends on individual projects.

The most cost-effective and failure-detecting configurations vary based on the projects. We observe that the configuration "CPU 2 and RAM 4" is cost-effective for flaky test detection on many projects but it is never the best configuration for detecting flaky failures in a given project. We observe that the configuration "CPU 0.1 and RAM 1GiB" is best for detecting test failures in many projects, but is rarely most cost effective. The configuration "CPU 0.5 and RAM 2GiB" combines the desirable properties of the previously discussed configurations for our dataset and provides the highest number of flaky test detections for many projects, while also offering the best cost-effectiveness for many other projects. Coincidentally, this configuration ranked most commonly as "best price and detection" *and* for "best price and reliability," but *not* for the same projects.

Our experiments confirmed the presence of known flaky tests, and also detected some that were not previously reported in research literature. Considering the 30 open-source Java projects that we studied, we identified 256 flaky tests, of which 154 had been reported in prior literature [31], [43] and 102 that have not. Of the 128 flaky tests in these projects that we confirmed as RAFT, 75 had been reported as flaky in prior literature, and 53 had not. These Java projects have been studied far more extensively (re-run thousands of times by multiple research groups) than the Python and JavaScript projects (re-run dozens or hundreds of times by a single group). Considering the 10 open-source JavaScript projects in which we detected 92 flaky tests, prior research [36] did not detect any flaky tests. Considering the 12 open-source Python projects in which we detected 260 flaky tests, 9 were detected in prior research [34], and 251 were not reported in prior literature.

Summary: The most cost-effective configuration to detect RAFTs largely depends on the project.

V. Discussion

Our study confirms the presence of resource-affected flaky tests (RAFTs) in open-source Java, JavaScript, and Python projects. Rather than spend time trying to repair RAFTs, developers can immediately and directly reduce their failure rate by increasing the resources available when running them. Our experiments find that the presence and impact of RAFTs can vary substantially between projects. While some of these failures might be obvious—in the case of `incubator-dubbo`, we find tests that failed in almost every single run when executed with restricted resources—other tests have a more subtle dependency on system resources. As test suites grow and resources are increasingly stretched thin to run more test suites concurrently, developers should be aware of slowly increasing flaky test failure rates.

Projects with mostly small, deterministic tests are less likely to be impacted by resource-related flakiness than projects with large, resource-dependent integration tests. By specifying the expected resource requirements for reliably running tests, developers can reduce the occurrence of RAFT-related failures. In this section, we describe our experiences reporting these concerns to developers, provide a qualitative discussion of an exemplar non-RAFT flaky test, and discuss implications for future research in flaky tests.

A. Feedback from developers

To gain further insight into the implications of our study, we contacted developers of projects in which we detected RAFTs and suggested they update the project specifications (e.g., `README.md`) with the minimal resource configurations that should be used to mitigate RAFTs. As developers are likely more hesitant to accept changes related to their projects' specifications (than typical, code-related changes), we open issue reports on only a subset of our evaluation projects to gauge developers' interest in our findings—specifically, we

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3462251

Table IV: Average test suite execution time (seconds) under the baseline 4CPU, 16GiB RAM configuration and average seconds slowdown (positive values) or speedup (negative values). Bolded values indicate the cheapest, most reliable configuration for that project.

			CPU 0.1		CPU 0.25	CPU 0.5		CPU 1		CPU 2			CPU 4
	Project	Baseline	1G RAM	2G RAM	2G RAM	2G RAM	4G RAM	4G RAM	8G RAM	4G RAM	8G RAM	16G RAM	8G RAM
Java	assertj-core	8.56	227.43	225.22	76.68	32.15	31.59	11.04	10.68	1.42	**1.34**	1.33	-0.15
	carbon-apimgt	27.50	130.53	129.03	**43.13**	18.04	17.74	5.99	5.80	0.64	0.63	0.63	-0.03
	commons-exec	55.77	68.60	68.52	22.48	8.77	8.69	2.45	2.43	**0.12**	0.11	0.16	0.00
	db-scheduler	13.98	81.48	79.50	26.76	10.85	10.70	3.50	**3.44**	0.25	0.22	0.27	-0.04
	delight-nashorn-sandbox	18.15	464.71	440.55	140.98	58.55	55.97	18.43	18.23	0.36	0.98	1.45	**-0.57**
	elastic-job-lite	40.15	356.55	348.77	106.24	**41.22**	39.35	12.00	11.99	0.84	0.81	0.92	0.03
	esper	9.15	95.53	94.48	**31.68**	12.99	12.71	4.29	4.21	0.46	0.44	0.44	0.01
	excelastic	7.84	85.22	82.09	**27.71**	11.37	11.23	3.60	3.63	0.25	0.21	0.19	-0.05
	fastjson	34.61	534.06	530.96	178.63	73.43	73.63	23.24	21.33	**2.25**	2.32	3.49	-0.54
	fluent-logger-java	35.24	97.20	94.77	31.30	**12.90**	12.64	4.08	3.82	0.58	0.58	0.34	0.27
	handlebars.java	7.67	168.44	166.77	56.76	**23.98**	23.52	8.02	8.00	0.78	0.74	0.78	-0.02
	hector	32.39	537.63	528.89	176.08	69.14	68.72	19.57	18.72	0.06	**0.18**	0.12	0.02
	http-request	3.07	71.67	68.96	22.97	**9.49**	9.33	3.11	3.02	0.25	0.24	0.23	0.02
	httpcore	13.40	142.43	137.68	45.04	18.36	17.96	5.37	5.16	**0.34**	0.32	0.33	-0.01
	hutool	2.49	61.16	60.60	20.37	8.63	8.36	2.86	2.86	**0.27**	0.27	0.24	0.01
	incubator-dubbo	1,472.56	167.72	164.48	**52.56**	25.12	23.70	7.08	7.11	0.98	1.09	0.84	0.01
	java-websocket	10.35	121.45	**122.69**	42.49	16.99	16.50	5.73	5.63	0.71	0.66	0.69	-0.05
	logback	223.59	91.49	101.92	19.57	9.13	22.15	10.35	10.74	0.76	**0.87**	0.59	-0.19
	luwak	11.07	164.87	166.13	52.35	21.56	20.86	**7.02**	6.72	0.60	0.61	0.66	-0.01
	ninja	17.80	335.01	328.09	109.23	44.97	43.78	14.45	14.64	**0.73**	0.67	1.13	0.09
	noxy	18.01	361.16	356.28	122.18	**50.79**	49.92	16.94	16.80	1.80	1.55	1.80	0.09
	orbit	10.31	122.09	126.11	42.26	17.41	17.60	5.90	5.73	0.78	0.72	**0.61**	-0.04
	oryx	28.74	**110.11**	110.13	36.44	14.87	14.55	4.91	4.86	2.49	2.48	2.49	-0.01
	riptide	15.15	171.60	169.53	59.47	**29.88**	29.66	16.63	16.54	10.67	10.68	10.67	0.01
	rxjava2-extras	49.65	270.56	272.01	90.07	**35.47**	35.28	10.08	9.65	1.74	1.62	2.01	-0.55
	spring-boot	91.53	1,640.48	1,621.49	544.73	217.67	216.10	72.82	71.24	4.87	**4.88**	4.92	-0.97
	timely	11.15	328.17	314.87	107.61	43.95	42.51	14.96	14.52	**1.87**	1.91	2.01	-0.05
	wro4j	74.69	3,319.11	2,000.16	726.08	322.79	**310.57**	119.61	114.43	28.17	23.26	22.54	0.41
	yawp	7.36	111.09	111.62	36.81	15.16	14.92	5.04	4.98	**0.54**	0.55	0.56	0.02
	zxing	66.83	764.28	765.47	251.67	89.36	89.33	10.29	**10.41**	0.70	0.57	0.71	0.09
JavaScript	apollo-client-devtools	21.14	-	836.01	239.46	98.62	123.91	47.37	40.90	**11.10**	11.14	11.15	-0.19
	IcedFrisby	15.56	4.21	4.14	**1.09**	0.25	0.12	0.02	-0.05	-0.02	-0.01	-0.02	-0.03
	javascript-action	852.01	**2,944.00**	3,437.00	1,069.00	431.00	252.00	25.00	35.00	-1.00	1.00	-3.00	-4.00
	ngrok	6.43	**14.36**	14.17	3.78	0.68	0.54	-0.44	-0.58	-0.55	-0.75	-0.80	-0.51
	preset-modules	8.95	356.16	336.12	132.46	57.65	54.27	21.80	22.10	**5.82**	6.20	6.12	-0.14
	react-datetime	3.39	16.13	15.29	5.11	**1.95**	1.98	0.46	0.46	0.05	0.02	0.02	-0.01
	react-native	59.16	-	1,761.38	745.77	335.32	324.85	114.02	113.28	**25.84**	25.53	27.52	1.47
	shields	0.82	10.51	10.61	**3.63**	1.35	1.34	0.26	0.24	-0.01	0.00	-0.01	0.00
	tippyjs-react	4.95	213.68	207.47	73.10	31.65	30.33	12.54	12.17	**3.48**	3.51	3.49	0.01
	twilio-video-app-react	86.01	-	-	541.41	224.16	139.64	**28.91**	160.25	2.70	33.22	37.13	-0.01
Python	celery	111.36	**6.15**	6.15	6.15	6.15	82.70	-0.67	0.65	-0.17	-0.65	-0.22	-1.26
	conan	690.56	6,159.97	5,961.89	1,879.60	**591.21**	590.22	7.99	12.65	4.76	8.96	-1.86	0.09
	electrum	83.35	657.47	658.99	209.34	**62.13**	66.87	0.35	0.84	0.33	0.44	1.37	0.31
	fonttools	28.18	269.92	268.89	87.27	**28.64**	28.75	0.16	0.08	-0.26	0.01	-0.08	0.07
	ipython	53.83	277.28	274.08	85.94	**28.43**	28.33	0.62	0.46	-0.18	-0.04	0.11	-0.09
	loguru	82.52	301.01	355.34	95.33	**28.10**	30.03	-0.10	1.16	0.60	-0.08	0.08	-1.30
	mitmproxy	26.81	208.42	203.08	66.69	**21.67**	21.91	0.10	-0.02	-0.06	0.08	0.02	0.03
	requests	103.03	18.89	19.62	6.39	1.94	1.78	**-0.07**	-0.02	0.02	-0.04	-0.14	-0.18
	seaborn	526.53	-	3,602.62	v1,295.04	580.38	580.52	189.93	193.15	47.45	48.64	50.52	-0.85
	setuptools	857.70	3,746.12	**3,829.87**	864.35	40.80	36.10	5.34	5.06	0.38	-2.54	-3.62	-0.52
	sunpy	167.72	1,790.21	1,807.60	609.78	**224.48**	225.18	35.91	36.53	10.60	11.04	11.08	-0.50
	xonsh	80.92	818.52	828.30	257.04	**86.68**	84.42	2.23	2.02	-0.34	-0.23	0.22	0.70

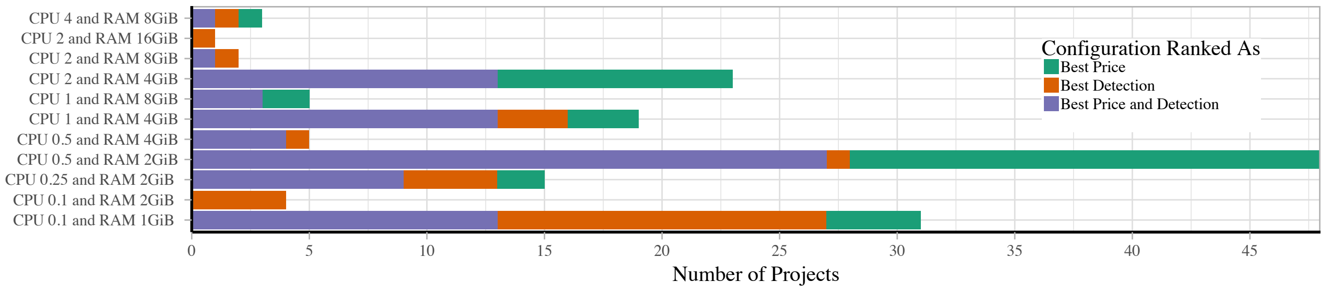


Figure 6: What are the best resource configurations to detect flaky failures? For each configuration that we analyzed, we show the number of times that it was best at detecting flaky tests (number of unique flaky tests detected), the best in terms of price, or the best in terms of both. If a configuration was tied for best in terms of detection for a project, we select the cheaper one. We hide configurations that were not optimal on either dimension.

start by engaging with developers only for our Java-based projects. We next describe the process by which we contacted developers, along with a summary of the developers' responses to our recommendations.

1) Overview: First, we identify Java projects from our evaluation that:

- *are active*, i.e., had a commit or developer interaction on issues or pull requests in the last three years (this resulted in only 18 of the 30 projects);
- *are runnable*, i.e., could be cloned at their latest version, the code compiles, and the tests run (all 18 ran);
- *had a flaky test in their latest revision*, i.e., we run the project test suites 300 times in each of the 12 throttling configurations from Table II, and keep the projects that have at least one RAFT (14 of the 18 projects had at least one RAFT). The latest version of each project that we used is in our artifact [23].

For each of the remaining 14 projects, we initiate communication with developers by creating an issue, either via GitHub or on the project's custom issue tracking system, that clearly indicates which tests are RAFTs, how to reproduce the issue (with step-by-step instructions on how to clone the code, build the Docker container, and run the tests), and which configuration likely prevents RAFTs.

2) Feedback: For 14 projects that we initiated communication with, six have responded to us and the remaining eight are still pending with no response. In two of the six projects [44], [45], developers quickly asked for or created their own PRs with minimum required resources to avoid RAFTs after we reached out. In two other projects [46], [47], developers initially expressed confusion to the concept of RAFTs and did not understand why stating minimum requirements for running tests were relevant. However, after follow-up comments describing how the RAFT configurations were plausible cases in which a contributor might run their tests, the developers agreed with our concern and accepted our proposed documentation changes. Lastly, there were two projects [48], [49] in which developers felt that the change to documentation was not appropriate, noting that these tests might be flaky "regardless of resources" and preferred to create a task to improve those tests to reduce their flakiness directly.

3) Discussion: There were common threads throughout our interactions with developers.

First, developers generally want to reduce the flakiness in their test suites, but are more focused on fixing tests to make them less flaky. In `http-core` [46], [50], the developer was initially quite dismissive, and eventually acquiesced with the caveat that they would "rather see efforts spent on analyzing the failing test cases and fixing them". The developer in `shardingsphere-elasticjob` [49] simply linked our issue to another flaky-test related issue, and eventually fixed the flakiness by using more robust assertions. These developers were not convinced that poorly provisioned machines can lead to more flakiness, perhaps because they do not experience the flakiness on their own machines or believe that their application would be used on such machines.

In fact, developers of the `timely` [48] project had difficulties imagining their applications/test suites would be run in resource-constrained environments. The developer was not interested in specifying minimum resources because they said their application is "designed to work with Apache Accumulo, an inherently 'big data' application". Similarly, in `db-scheduler` [47], [51], the developer said that they always "run the tests on a multicore machine". In these situations, noting that different developers have different machine specifications helped. (Note that suggesting asynchronously executing test suites in cloud environments may also have convinced developers, though we did not try this suggestion.)

There are also developers like those in `delight-nashorn-sandbox` [45] and `dubbo` [44] that were immediately interested in our fixes. In fact, the developer of the former created a new test that failed if system specifications were not adequate to better inform developers that RAFT failures may be occurring. This example highlights how developers may be content with simple RAFT mitigation strategies.

In summary, we reached out to the developers of 14 Java projects regarding RAFTs and the minimum machine specifications their projects' should run in to avoid RAFTs. Of the 14 projects we reached out to, developers of six projects have responded to us, while the remaining eight are pending with no response. Of the six that responded, developers of four projects have improved their project based on our suggestions, while the remaining two indicated that they prefer actual fixes over specification clarifications.

B. Qualitative examination of flaky tests

To complement our statistical analysis, we also provide a qualitative discussion of exemplar an unusual non-RAFT flaky tests.

1) Non-RAFT flaky tests: Of course, not all flaky tests are RAFTs. We provide a qualitative discussion of two flaky tests that we observed in our dataset, which are not RAFTs. The `Timely` [52] project contains two tests from the class `TimeSeriesGroupingIteratorTest` that are flaky but non-RAFT: `testTimeSeriesDropOff` and `testMultipleTimeSeriesMovingAverage`. The goal of these tests is to check that the averages of numeric values in two data structures are the same. However, the tests use time-based random number generation, which results in unpredictable and unreliable test results.

The flakiness of these tests is not dependent on the environment in which they are run, but rather on the time in which they are run. A developer can run the same test multiple times, even in different environments, and get different results each time. In our experiments, we observed that these tests were flaky in all configurations. In the baseline configuration, the tests failed eight times in 300 runs, while in other configurations, the tests failed at least once and up to 12 times. Our findings for this test are confirmed by our prior anlaysis [53], which also described these tests to be flaky due to time. In fact, that prior analysis found a 2.6% failure rate for these tests, which translates to roughly eight failures in 300 runs.

C. Implications for researchers

Much interesting and important research can build on and extend this study. A notable limitation of this study is its reliance on relatively short-building, open-source projects. Future work should confirm the impact of RAFT in closed-source projects, and might also explore approaches to automatically triage and repair RAFT. Without such validation, it is challenging to evaluate the direct impact of flaky test research on the practice of software engineering.

Conducting such research can be challenging, as it requires *running* test suites, requiring industrial collaborators to provide access to a complete codebase and a suitable environment for running it. However, the complexity of the study may be worth the investment, as our anecdotal evidence from conversations with practitioners suggests that RAFT could be an even larger concern at some companies than in the open-source projects that we studied. Specifically, open-source projects with very limited budgets for continuous integration (e.g. relying on the limited free tiers of services) have a limited tolerance for long-running, frequently failing continuous integration pipelines. Companies that are racing to spend investor money and launch a product as soon as possible may have different incentives. Do companies with fast-release schedules and a healthy flow of revenue to pay the CI bills end up with flakier and longer-running test suites than the open-source projects that we studied? We believe that there is no single definitive answer to this question yet it strongly motivates future research and industrial collaborations.

This study also may have implications for other flaky test research. One line of flaky test research has focused on *detecting* flaky tests, generally by re-running them hundreds [24] or thousands [31], [53] of times to detect unlikely failures. We have found that these experiments can be conducted more cost-effectively by reducing the resources that are used for each test execution: providing each test suite with 4 CPUs and 16GiB of RAM may be an over-provisioning of resources. As these experiments can involve (re-)running a test suite tens of thousands of times, there may still be a noticeable improvement even for test suites that take under a minute to complete. From our experiments, we found that as long as there is enough RAM available to reliably complete a test suite execution without reaching a fatal out-of-memory error, reducing resources available to a test suite increases the number of flaky tests detected. Rather than deploying "stressor" tasks that acquire CPU and RAM in an effort to starve tests of resources [21], [29], it may be substantially cheaper to simply limit the resources available to those tests, and use those resources for other purposes.

Surveys of developers show that detection of flaky tests is a less pressing problem than the mitigation of flaky tests [54]. This article outlines a simple, yet effective approach for mitigating the impact of flakiness in test suites, when that flakiness is tied to resource availability. Researchers should investigate other approaches to reduce the incidence of flaky failures, considering factors beyond the test code itself, such as environmental factors. Our supplemental artifact includes our dataset and the scripts used to detect RAFTs from test executions [23].

D. Implications for Continuous Integration Infrastructure

Cloud-based continuous integration (CI) systems have become increasingly popular. Major cloud vendors provide CI services, such as Amazon's CodeBuild [55], Microsoft's Azure DevOps Pipelines [56] and Google Cloud Build [57]. We reviewed the pricing and configuration options available for popular cloud-based CI services, to see how the configurations aligned with the resource configurations that we evaluated. Specifically, we reviewed the configuration and pricing of Amazon's CodeBuild [55], Microsoft's Azure DevOps Pipelines [56], Google Cloud Build [57], GitHub Actions [58], GitLab CI/CD [59], BitBucket Pipelines [60], CircleCI [61], TravisCI [62] and TeamCity [63]. Builds are executed by *CI runners*, which may be provided by the cloud service ("cloud builders"), or managed by developers using their own ("self-hosted builders"). Some services provided only a single configuration of cloud builder: Azure DevOps and GitHub Actions both provide runners with 2 CPUs and 7GiB of RAM [56], [58] (at time of writing, GitHub has a beta-only feature to support larger cloud runners). BitBucket provides an unspecified CPU resource, but allows memory to be scaled between 4 and 32 GiB [60]. GitLab and Google Cloud Build allow developers to select as little as 1 CPU with 4GiB of RAM [57], [59], while Amazon CodeBuild, GitLab CI, TravisCI and TeamCity start at 2 CPUs with 4GiB of RAM, with a maximum configuration (on Amazon CodeBuild) of 72 CPUs and 144GiB of RAM [55], [59], [62], [63].

1) Potential cost savings: Our dataset is largely skewed to projects with relatively fast-running test suites (as shown in Table IV, the median test suite execution time is 52 seconds), and hence, it is not possible to draw strong conclusions regarding the potential cost savings, as the cost of running these test suites is already quite small. As shown in Figure 8, the median savings that we calculated was only $0.10 per month, with a maximum of just $0.48. However, we evaluate the potential for cost savings in these projects to provide motivation for researchers and practitioners to examine more significant cost savings on other projects. Our finding that some projects can reliably build with only 0.5 CPU and 2GiB of RAM indicates that some developers may be able to save money by using lower-end CI runners than are available using the "cloud runner" model. Each of these CI services *also* supports a "self-hosted" runner model, where builds take place on compute resources that are managed by the developers (e.g., a dedicated "builder" machine, or an auto-scaling cluster of builders). For example: developers could deploy an auto-scaling cluster of builders with 0.5CPU/2GiB RAM on AWS for $0.008739/hour, while GitHub Actions would charge about 55 times as much ($0.008/*minute*) for a runner with 2 cores and 7GiB of RAM. Furthermore, each of these services support only a limited number of resource configurations, forcing specific combinations of CPU and memory (e.g., on CircleCI, a configuration 1 CPU with 4GiB of RAM is not available, developers must pay for 2 CPUs to receive 4GiB of RAM). Based on the mismatch between the resource

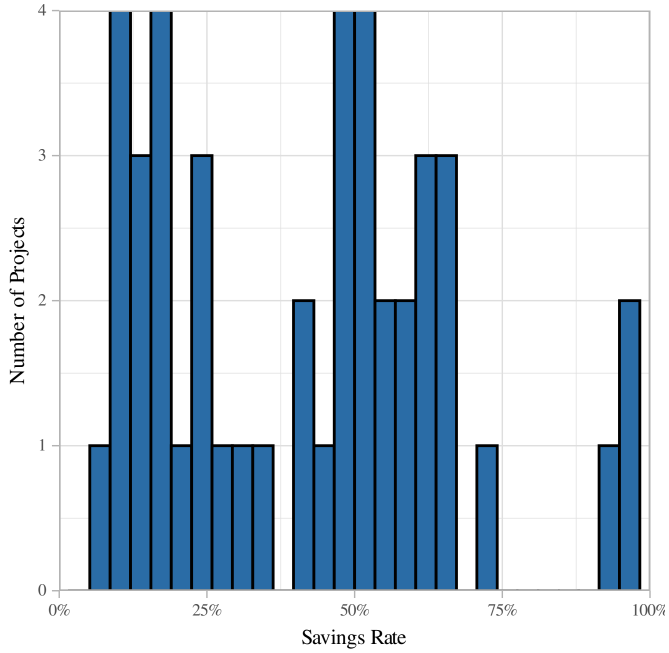


Figure 7: Cost savings rates per-project, simulating a change from a 2 CPU/8GB RAM configuration to the cheapest configuration that also minimized the number of flaky tests. We observe a savings in 44 projects. We noted that a greater resource configuration was needed to minimize flakiness in four projects, and that the baseline 2 CPU/8GB RAM configuration was optimal for four projects.

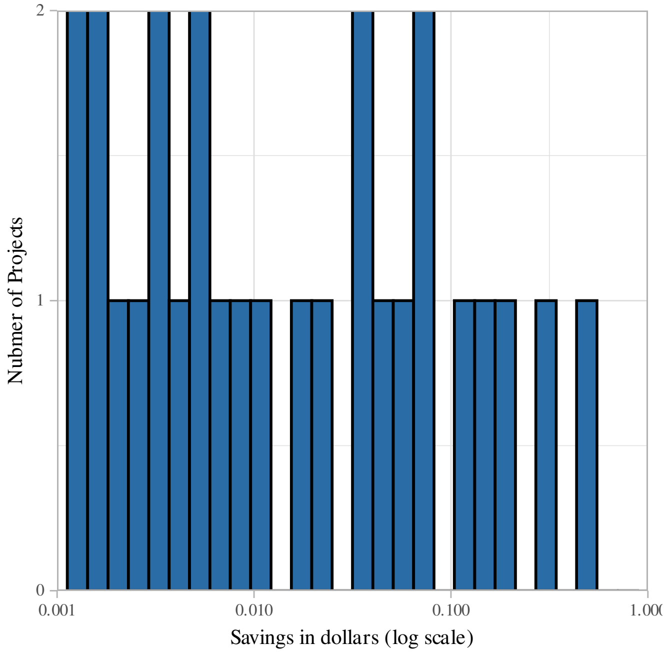


Figure 8: Expected monthly cost savings in USD per-project, simulating a change from a 2 CPU/8GB RAM configuration to the cheapest configuration that also minimized the number of flaky tests. As the test suites for many projects were quite short (median time = 52 seconds, Table IV shows execution times) and number of revisions per-month is low (median=29), the absolute cost savings are negligible.

requirements of projects and the configurations provided by cloud CI services, we speculate that significant cost savings may be achievable for some developers. Using "self-hosted" runners that auto-scale on containers [64] that match the actual resources required by a build (rather than over-provisioning) can have significant cost savings. Intuitively, the magnitude of the cost savings will be proportional to how long it takes to run the test suite: reducing the cost of a 5-hour test suite will have greater absolute cost savings than reducing the cost of a 52-second test suite. Similarly, the magnitude of the cost savings will vary with how frequently the test suite is executed.

The actual cost savings per-project will vary based on 1) the proportional reduction in resource costs per-hour, 2) the duration of each test suite and 3) the frequency with which each test suite is executed. We first explore the reduction in resource costs per-test suite execution, as this is the dependent variable that we study directly. We calculate the expected cost of running each test suite once under each configuration, and report the proportional savings between the "cheapest most reliable" configuration and the default 2-CPU 8GB RAM configuration. Figure 7 shows a histogram of cost savings per-project in our dataset. Of the 52 projects, we observed a cost savings in 44, no change in four, and an increase in costs in four projects (for which the 2 CPU configuration was not the most reliable).

Given the relatively short duration of each of the studied test suites (min=2 seconds, median=52 seconds,max=14 minutes), the absolute cost savings per-project are low. We calculate the expected monthly savings under the assumption that a CI build is executed exactly once per-commit. To find the average monthly savings, we compute the average number of commits per month, and multiply that times the expected cost per-test suite execution at each resource level. Figure 8 shows that the expected monthly savings per-project is at most $0.48. While the absolute savings may not be meaningful on these projects, we speculate that projects with longer-running test suites that receive more frequent commits will see greater absolute savings, as indicated by the relative savings shown in Figure 7.

We empirically demonstrate that reducing the number of CPUs available for running tests (particularly below 1 full CPU) has a significant effect on the test suite execution time. Our analysis provides insight into the relationship between CI resources, flakiness, and cost to help developers reason about these trade-offs in their own context.

2) Efficient Resource Utilization: A cost-savings may not be recognizable in contexts where CI builds are run on pre-allocated, dedicated resources. In these cases, understanding RAFT can help developers increase the utilization of those build resources while avoiding flakiness. For example, if it is determined that a frequently executed test suite could run with 2 CPU cores instead of 4, then the throughput of the CI pipeline can effectively be doubled by reducing the resources provided to each run. In order to realize a net gain in test suite throughput, it is also important to ensure that a reduction in resources does not increase the duration of each test suite execution more so than the benefit provided by added parallelism. Based on our analysis of slowdowns resulting from resource restrictions (shown in Table IV), we note that with the exception of projects with very fast-running test suites (e.g. under 60 seconds), the slowdown incurred by the resource reduction is often less than the potential gain in throughput from parallel runs.

3) Determining the ideal configuration: With the goal of exhaustively finding the most cost-effective resource configuration, developers should consider the kinds of operations that tests perform. While it might be tempting to assume that, for example, all database-related projects will be dependent on disk I/O more than CPU time (as database systems are typically I/O-bound), our qualitative analysis of RAFT suggests that configurations and resources are more related to the scope and features of tests rather than the scope of the overall project. A test suite validates concurrent or asynchronous behavior of a system and relies on timing is likely to be CPU-dependent, regardless of what the system under test does. Our quantitative analysis of the resources that affect flaky tests (RQ1) suggests that if developers only have time or interest in tuning a single resource to detect or avoid flakiness, they should tune CPU availability. We empirically demonstrate the importance of tuning these reduced resource configurations in order to mitigate the impact of flaky tests.

E. Threats to validity

1) Construct validity: There are two central constructs to our study, the flakiness of tests and resource-dependency.

Test flakiness: For classifying a test as flaky, we rely on the observation of different test outcomes across repeated executions. These executions may be affected by uncontrolled factors and, hence, we may erroneously classify tests as flaky. More precisely, there may exist an execution environment under which the tests do not non-deterministically pass and fail. However, the manifestation of this behavior demonstrates that the investigated tests *can* be flaky in *some* execution environment. This interpretation of flakiness follows common practice in existing work.

Resource dependency: In our work, we determine resource dependency by controlling resource access via Linux control groups in a uniform manner, i.e., resource accesses are affected throughout test execution. This access control closely resembles execution on a resource-constrained machine. It does not resemble resource constraints resulting from dynamic load on shared resources well, as they are proposed in Terragni et al.'s work [29]. However, the extreme resource restrictions we experimented with for Phase I of our work resemble extreme dynamic loads and lead to very sensitive detection. We consider that part of our experiments to be a desirable property for a detector of rare events like flaky test failures. For Phase II, the uniform resource restriction leads to potentially optimistic results if control for additional load on the test setup cannot be controlled for, which is an important restriction that users of our results should be aware of.

2) Internal validity: The conclusions we draw are based on 300 re-executions of tests under each configuration. Other work has shown that flaky tests can fail much more infrequently than once in 300 runs [31] and our results do not systematically address such rare cases. The focus of our work lies on flaky tests that fail frequently enough to significantly disrupt developer activity.

To classify tests as RAFTs, we rely on a χ^2 test of independence between failure rates under normal operation and resource constraints. As we consider different configurations for analyzing the effect of resource constraints, we conduct several such tests against the same baseline failure rate, which may lead to multiple comparison problems. We account for these by adjusting the obtained p-values using the Benjamini-Hochberg procedure [65].

3) External validity: Our results are restricted to the studied projects and may not be generalized to other projects. One particular concern regarding generalizability centers on how long each test suite takes to run. As noted throughout this article, the absolute cost savings that developers might witness by optimizing resource utilization is bounded by how long the test suite typically takes to run. It would indeed be challenging to witness significant absolute cost savings for most of the projects that we examined in our dataset, as the total cost of running the test suite is quite low to begin with (the median test execution time is just 52 seconds). We assume that the cost of running a test suite is proportional to both the cost-per-second of a computing resource and the duration of a test suite. Furthermore, we assume that it is possible to vary the resources that are provided to a particular test suite — some execution environments might simply provide a fixed set of resources to all test suites.

We expect that the main conclusions of this paper (that RAFTs exist, and that failure rates of RAFTs can be influenced by adjusting resource constraints) will hold. However, it would be difficult to extrapolate from our study to determine precisely how prevalent RAFTs are in software overall or which resource configurations are the "best" overall for reducing or increasing those failure rates. Even from our study of only 52 open-source projects, we can see that the observed prevalence of flaky tests and their sensitivity to resource restrictions differ. We, hence, recommend reassessing RAFTs for other projects using the methodology outlined in this work.

VI. Conclusions

Using rigorous statistical methods, we have empirically demonstrated the link between test flakiness and the resources available for running tests. Our study of 52 Java, JavaScript, and Python open-source projects revealed that resource-affected flaky tests (RAFTs) may be more prevalent in some projects than others, likely tied to the kinds of behaviors that each projects' test suite examines. By controlling the quantity of CPU cores and RAM available to a test suite while it runs, developers can reduce the likelihood of observing flaky failures, or if desired, increase it. When we reached out to the developers of 14 projects regarding the minimum machine specifications their projects' should run in to avoid RAFTs, developers of six projects responded, while the remaining eight are pending with no response. Of the six that responded, developers of four projects improved their project based on our suggestions, while the remaining two indicated that they prefer actual fixes over specification clarifications. Although the test suites that we studied were relatively fast running (median=52 seconds), we examined the potential for cost savings, hypothesizing that a similar relative cost savings would apply to other projects where the absolute savings could be more significant. Future research in RAFTs should examine their prevalence and impact on industrial projects. Other work in detecting flaky tests will benefit from running tests in reduced resource configurations, which may be cheaper to run and reveal more flaky failures. Future research on RAFTs might consider examining (1) the different failures that occur under different resource configurations for a given test, (2) the impact of other environmental factors on flaky-test failures, (3) the idea of ignoring test runs when there are insufficient resources to reliably run tests, and (4) how regression testing techniques, such as test parallelization, can leverage RAFT information to allocate machines for testing.

Acknowledgements

This work was supported in part by the US National Science Foundation under grants NSF CCF-2100037, CCF-2319472, CCF-2338287, CCF-2349961, CNS-2100015, by the Brazilian National Council for Scientific and Technological Development (CNPq) under grant 140220/2022-4, by the Deutsche Forschungsgemeinschaft (DFG) 496588242 (IdeFix), and the LMU PostDoc Support Funds. We also acknowledge support for research from Dragon Testing.

References

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

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

[3] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *TOSEM*, 2021.

[4] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *ICSE SEIP*, 2020.

[5] J. Malm, A. Causevic, B. Lisper, and S. Eldh, "Automated analysis of flakiness-mitigating delays," in *AST*, 2020.

[6] M. H. U. Rehman and P. C. Rigby, "Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson," in *ESEC/FSE Industry Track*, 2021.

[7] "Facebook testing and verification request for proposals," Accessed 2023. [Online]. Available: https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019

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

[9] Google, "TotT: Avoiding flakey tests," Accessed 2023. [Online]. Available: http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html

[10] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *ICSE SEIP*, 2017.

[11] J. Micco, "The state of continuous integration testing at Google," in *ICST*, 2017.

[12] C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale," in *ICSE*, 2017.

[13] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing," in *ICSE*, 2017.

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

[15] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE*, 2015.

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

[17] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE*, 2020.

[18] T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests," in *ESEC/FSE*, 2022.

[19] "Test verification," Accessed 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

[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.

[21] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with Shaker," in *ICSME*, 2020.

[22] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," ser. FSE 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[23] D. Silva, M. Gruber, S. Gokhale, E. Arteca, A. Turcotte, M. d'Amorim, W. Lam, S. Winter, and J. Bell, "The Effects of Computational Resources on Flaky Tests (Artifact)," https://zenodo.org/doi/10.5281/zenodo.13711427, Oct. 2023.

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

[25] 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.

[26] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[27] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in Python," in *ICST*, 2021.

[28] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of practitioners' perceptions of test flakiness factors," *Software Testing, Verification and Reliability*, vol. 31, no. 8, p. e1791, 2021.

[29] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *ICSE NIER*, 2020.

[30] "Java WebSockets," Accessed 2023. [Online]. Available: https://github.com/tootallnate/java-websocket

[31] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *ICSE*, 2021.

[32] 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: 31st IEEE International Conference on Software Reliability Engineering*, Virtual Event, October 2020, pp. 403–413.

[33] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Evaluating features for machine learning detection of order- and non-order-dependent flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 93–104.

[34] ——, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empir. Softw. Eng.*, vol. 28, no. 3, p. 72, 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10307-w

[35] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test Flakiness Across Programming Languages," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2039–2052, 2022.

[36] G. A. Yost *et al.*, "Finding flaky tests in JavaScript applications using stress and test suite reordering," Ph.D. dissertation, 2023.

[37] E. Arteca and A. Turcotte, "Npm-filter: Automating the mining of dynamic information from npm packages," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 304–308. [Online]. Available: https://doi.org/10.1145/3524842.3528501

[38] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[39] Amazon, "Aws fargate," 2023. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/fargate.html

[40] Google, "Google kubernetes engine (gke)," 2023. [Online]. Available: https://cloud.google.com/kubernetes-engine

[41] Microsoft, "Azure kubernetes service (ake)," 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/kubernetes-service

[42] Amazon, "Aws fargate pricing," 2023. [Online]. Available: https://aws.amazon.com/fargate/pricing/?nc1=h_ls

[43] W. Lam, "Illinois Dataset of Flaky Tests (IDoFT)," Accessed 2023. [Online]. Available: http://mir.cs.illinois.edu/flakytests

[44] Agorguy, "dubbo pr-12391," 2023. [Online]. Available: https://github.com/apache/dubbo/pull/12391

[45] ——, "delight-nashorn-sandbox issue-137," 2023. [Online]. Available: https://github.com/javadelight/delight-nashorn-sandbox/issues/137

[46] ——, "http-core issue-747," 2023. [Online]. Available: https://issues.apache.org/jira/browse/HTTPCORE-747

[47] ——, "db-scheduler issue-388," 2023. [Online]. Available: https://github.com/kagkarlsson/db-scheduler/issues/388

[48] ——, "timely issue-242," 2023. [Online]. Available: https://github.com/NationalSecurityAgency/timely/issues/242

[49] ——, "shardingsphere-elasticjob issue-2219," 2023. [Online]. Available: https://github.com/apache/shardingsphere-elasticjob/issues/2219

[50] ——, "http-core pr-410," 2023. [Online]. Available: https://github.com/apache/httpcomponents-core/pull/410

[51] ——, "db-scheduler pr-403," 2023. [Online]. Available: https://github.com/kagkarlsson/db-scheduler/pull/403

[52] "Timely," Accessed 2023. [Online]. Available: https://github.com/NationalSecurityAgency/timely

[53] 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.

[54] M. Gruber and G. Fraser, "A survey on how test flakiness affects developers and what support they need to address it," in *ICST*, 2022.

[55] Amazon Web Services, "Aws codebuild pricing," https://aws.amazon.com/codebuild/pricing/, 2023.

[56] Microsoft, "Pricing for azure devops," https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/, 2023.

[57] Google Cloud Build, "Cloud build pricing," https://cloud.google.com/build/pricing, 2023.

[58] GitHub, "About github-hosted runners," https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners, 2023.

[59] GitLab, "Saas runners on linux," https://docs.gitlab.com/ee/ci/runners/saas/linux_saas_runner.html, 2023.

[60] Atlassian, "Bitbucket support - runners," https://support.atlassian.com/bitbucket-cloud/docs/runners/, 2023.

[61] CircleCI, "Configuring circleci," https://circleci.com/docs/configuration-reference/, 2023.

[62] TravisCI, "Travisci - build environment overview," https://docs.travis-ci.com/user/reference/overview/, 2023.

[63] JetBrains, "Teamcity cloud," https://www.jetbrains.com/teamcity/cloud/, 2023.

[64] GitHub, "Autoscaling with self-hosted runners," https://docs.github.com/en/actions/hosting-your-own-runners/autoscaling-with-self-hosted-runners, 2023.

[65] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995. [Online]. Available: http://www.jstor.org/stable/2346101

[66] Accessed 2023.

[67] 2019.

[68] *ICSE 2015, Proceedings of the 37th International Conference on Software Engineering*, 2015.

[69] *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*, 2017.