DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Reducing Effort for Flaky Test Detection Through Resource Limitation

Maximilian Schallermayer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Reducing Effort for Flaky Test Detection Through Resource Limitation

Reduzierung des Aufwandes der Erkennung von Flaky Tests durch Limitierung von Ressourcen

Author:	Maximilian Schallermayer
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	Fabian Leinen
Submission Date:	April 15, 2023

I confirm that this bachelor's thesis in information systems is my own work and I have documented all sources and material used.

Munich, April 15, 2023 Maximilian Schallermayer

Abstract

In software engineering, flaky tests are tests that exhibit inconsistent behavior, meaning they produce different outputs when executed multiple times under the same conditions. This behavior causes problems during regression testing, where each failure is supposed to indicate a bug introduced by the latest revision to the code base.

A common strategy to detect flaky tests is *Rerun*, which executes each test repeatedly under the same conditions. The test is marked as flaky if both passing and failing results are observed. However, since flaky tests often only rarely fail while passing most times, *Rerun* is inefficient as it may take many runs to detect a test as flaky.

We present a more efficient approach that focuses on detecting concurrency-related flakiness by limiting computing resources through Docker. Besides trying to detect flaky tests more quickly, we also examine how the limitation affects previously stable tests. As we aim to detect those tests that are most likely to cause problems in practice, flagging stable tests as flaky is undesirable as it may divert developers' attention away from the more critical cases of flakiness.

We achieve promising results as our approach reduces the number of runs needed to detect flakiness by almost 80%. When looking at absolute run time, we can provide a speedup of about 50%. However, we also observe that some previously stable tests fail under resource limitations. We find that clearly differentiating between flaky and stable tests proves challenging.

Contents

Abstract **iii**

1. Introduction **1**

2. Foundations **3**
2.1. Definition of Flaky Tests 3
2.2. Issues Caused by Flakiness 3
2.3. Root Causes of Flakiness 4
2.4. Severity of Flakiness 6
2.5. Limiting Resources Through Docker 6

3. Related Work **7**
3.1. Root Causes 7
3.2. Detection Strategies 7
3.3. Metrics for Severity 8
3.4. Fixing Flaky Tests 9
3.5. Research Gap 9

4. Approach **11**
4.1. Classifying Tests into Flaky and Stable 11
4.2. Connecting Computing Resources and Flakiness 12
4.3. Running Tests with Varying Resource Limitations 13
4.4. Measuring Configuration Performance 16

5. Evaluation **17**
5.1. Research Questions 17
5.2. Study Subject 18
5.3. Setup for Evaluation 19
5.3.1. Evaluation Metrics 19
5.3.2. Evaluation Baselines 20
5.4. Results 21
5.4.1. Accuracy of Detection 25

5.4.2. Reduction of Detection Effort 26
5.5. Discussion 28
5.5.1. Assessing the Reproducibility of Flaky Failures 28
5.5.2. Analysing the Low Precision 29
5.5.3. Future Work 30

6. Threats to Validity **32**
6.1. Threats to External Validity 32
6.2. Threats to Internal Validity 32

7. Conclusion **34**

A. Appendix **36**
A.1. Artemis Flaky Tests Strict 36
A.2. Artemis Flaky Tests Inclusive 36
A.3. Performance of Configurations 39

List of Figures **40**

List of Tables **41**

Acronyms **42**

Bibliography **43**

1. Introduction

Automated regression testing has become a vital part of modern software engineering. Changes to the codebase trigger the execution of a test suite, and developers expect failures to indicate newly introduced bugs. This workflow requires a deterministic behavior of tests, which is not always the case in practice. Tests that non-deterministically pass and fail when executed multiple times under the same conditions are called *flaky tests*.

The problems caused by flakiness are manifold. They can delay automatic integration and deployment, cause developers to waste time searching for non-existent bugs, and diminish the trust in the test suit, which may lead to developers ignoring tests and overseeing actual bugs [1].

A common approach to detect flakiness is to run tests repeatedly for a pre-defined number of times under the same conditions. The test is marked as flaky if both passing and failing results are observed. However, it can take many attempts before a test shows a flaky failure, which makes this approach rather time- and cost-intensive. For example, Gruber et al. found that around 170 runs are necessary to detect 95% of flaky Python tests [2]. Furthermore, prior work by Parry et al. suggests that the *Rerun* approach has limited effectiveness [3].

In order to make the detection process more efficient, some approaches rely on the root causes of flakiness. Root causes are certain conditions in the underlying test code that lead to flaky behavior. For example, Luo et al. found that the three main reasons for flakiness were *concurrency*, *asynchronous wait* (a sub-category of *concurrency*) and *test order dependency* [4]. Romano et al., who looked specifically at user interface tests, confirm that the *asynchronous wait* category makes up most of the flaky tests [5].

Due to how many flaky tests are caused by concurrency, we focus on detecting this specific category with our approach. We draw a connection between concurrency-related flakiness and computer resources such as CPU and main memory. We hypothesize that limiting the available resources can influence the order of events between threads and consequently increase the likelihood of flaky failures. We find an optimal limitation by repeatedly running different configurations and comparing the test results. We aim to

maximize the flake rate of flaky tests and minimize the number of failures of stable tests. We classify tests into flaky and stable in an up-front analysis of historical test results. Our goal is to provide a detection approach more efficient than regular *Rerun*. While prior work has already investigated the connection between concurrency-related flakiness and computer resources [6], our approach differs in three ways:

Firstly, we aim to detect practically relevant flakiness. We achieve this by analyzing which tests showed flaky behavior in the past and then maximizing the flake rate of flaky tests while minimizing the number of failures for stable tests. We hypothesize that this approach helps avoid overly restrictive resource limitations, which would generate results of little practical relevance.

Secondly, instead of putting real stress on the system by creating stressor tasks, we *limit* the resources through Docker. This approach may be more economical and manifest different flakiness due to the mechanism used to interfere with resource usage.

Thirdly, while previous work mainly studied flakiness regarding unit tests, we carried out our experiments on end-to-end tests. These tests are more complex and have longer average run times, and therefore present a promising study subject for efficient detection of flakiness.

To evaluate our approach, we first analyze the ratio between correctly detected flaky tests and falsely flagged stable tests. We then examine which kind of resource yields the most promising results and how much we can reduce detection effort. With the best configuration, we detect flaky tests with a recall of 77%, which is significantly higher than the baseline of 34% for weak limitations. We can reduce the number of runs needed to detect flakiness by almost 80% and achieve a speedup of about 50% for absolute runtime. Configurations targeting the CPU yield better results than those focusing on the main memory, especially when applied to the server instead of the client process.

We also find that the number of falsely flagged stable tests is substantially high, resulting in a precision of 40%. Through manual comparison of these failures with historic results, we find that some tests we classified as stable show flaky behavior on code versions different from the one used for our experiments. Therefore, some of the tests we classified as stable may actually be flaky, and the associated failures could negatively impact the precision of our configurations.

2. Foundations

In this chapter, we want to provide the necessary context in the field of flakiness required for the rest of this thesis. We start by defining flakiness and elaborating on the importance of the problem. We then provide a brief overview of the most common causes of flakiness and explain how the severity of flakiness varies between tests. We end by explaining on a high level how to limit computing resources through Docker.

2.1. Definition of Flaky Tests

A consensus on a uniform definition for flakiness has yet to be reached in literature [7]. Instead, practitioners often come up with slightly modified definitions to suit the context of their work. Most definitions have in common that a flaky test shows inconsistent results across multiple runs while controlling for certain factors to stay identical.

In this thesis, we will use the definition by Harman et al., who state that a *"flaky test is one for which a failing execution and a passing execution are observed on two different occasions yet for both executions, all environmental factors that the tester seeks to control remain identical"* [8]. The factors we seek to control are the code version for both the tests and the System Under Test (SUT) and the maximum amount of resources the testing processes can use.

2.2. Issues Caused by Flakiness

Flaky tests have become a frequent and crucial problem in automated regression testing [1, 9, 10]. In the following, we give a brief, non-exhaustive overview of some common issues caused by flakiness.

Many software projects utilize the concept of Continuous Integration (CI) and Continuous Delivery (CD). Each commit to the codebase triggers an automated build of the application and subsequent execution of the test suite. In case of successful execution, the changes can be directly deployed into production. A failure of the test suite implies

that the corresponding commit introduced a bug that must be fixed. However, if the test suite contains flaky tests, it might fail even though the changes introduced by the commit work as intended, delaying the deployment unnecessarily.

Debugging the flaky failure requires time. For once, the developer might look for the issue in the most recent commit to the SUT even though the reason for the failure resides in the test code [11]. Furthermore, replicating flaky failures in the developer's local environment is often difficult, increasing the required debugging time even further [12].

Another issue caused by flaky tests is the decreased confidence in the test suite [1]. When flaky failures become frequent, developers might resort to ignoring test results. Consequently, real bugs introduced by regression could go unnoticed and get released into production.

Lastly, the mitigation strategies introduced to combat flakiness cost companies a lot of time and money. For example, Google spends between 2% and 16% of its computing resources on re-running flaky tests [10].

2.3. Root Causes of Flakiness

Tests can produce a flaky failure for several reasons. Luo et al. provide ten categories for these root causes, with three of them making up the majority of cases: *concurrency*, *asynchronous wait*, and *test order dependency* [4].

Concurrency

Multiple threads interacting with each other can result in a non-deterministic order of events. If a test implicitly assumes one specific order, flaky behavior may occur. For example, a race condition arises if access to a variable shared between threads is not synchronized correctly. The final state might differ depending on which thread accesses the variable faster. Tests that expect the variable to be in one specific state will pass or fail respectively, depending on the actual state. It should be noted that non-determinism between threads may be a correct behavior of the SUT. However, in such cases, the test must account for all possible orders of events to produce consistent results.

Asynchronous Wait

Tests fall into this category if they make an asynchronous call and do not wait properly for the result to become available. For example, a test could request a resource from a remote service. If there is no proper synchronization to wait for a reply before trying to access the resource, the test can pass or fail, depending on whether the reply happened in a timely manner. Often, these tests include fixed time delays instead of explicitly waiting for a condition to be met, like receiving a reply from the server. While *asynchronous wait* is a subcategory of *concurrency*, it is often listed as a separate category by practitioners due to the large portion of tests that belong to it.

Test Order Dependency

Tests that produce different results depending on their order of execution within the test suite are called Order-Dependant (OD) flaky tests. If implemented following best practices, tests within a suite should be isolated and independent of each other so that the order in which they are executed does not influence the result. This isolation is important as many testing frameworks do not guarantee that tests are run in a specific order. For example, JUnit runs tests in a deterministic but not predictable order [13]. Nonetheless, test results in practice often depend on the order of execution. Tests may share state such as a field in main memory or some external resource like a database. Flakiness can arise if the state is not correctly set and cleaned up for each test individually.

Apart from these three categories Luo et al. specify seven more categories: *resource leak*, *network*, *time*, *IO*, *randomness*, *floating point operations* and *unordered collections*. Other practitioners have further expanded the set of root causes [1, 14].

While relatively efficient detection strategies exist for OD flaky tests [15–17], Non-Order-Dependant (NOD) flaky tests prove to be more challenging. For this reason, some practitioners treat the distinction between OD and NOD as a kind of higher-level categorization, with all the previously mentioned categories being a subcategory of NOD flaky tests [2].

Our approach aims to provide an efficient technique to detect NOD flaky tests of the *concurrency* and *asynchronous wait* categories.

2.4. Severity of Flakiness

In practice, most tests show flakiness to some degree, even if implemented following best practices [18]. However, the extent to which they exhibit flaky behavior differs between tests. In chapter 3.3, we will show how previous practitioners have developed metrics to measure the severity of flakiness. For our purposes, we will use the *flake rate* to determine the severity of flakiness, calculated by dividing the number of flaky failures by the total number of runs for a test.

Especially for larger test suites, it can be challenging to try fixing every test that exhibits flakiness. Instead, developers may focus their attention on the flakiest tests. We assume that tests with a high flake rate are more likely to cause problems during development, while stable tests with a low flake rate are less likely to be of issue. We will build on this assumption in chapter 5 to decide whether our approach detects practically relevant flakiness.

2.5. Limiting Resources Through Docker

Docker is a popular containerization platform that allows developers to bundle application code with the required dependencies and environment to provide efficient and portable deployment. Among many other features, it provides an API that lets the user specify the maximum amount of resources a given container can use [19]. To implement this feature, Docker uses Control Groups (cgroups) under the hood, a Linux kernel mechanism that allows the user to assign processes to hierarchical groups. The user can then configure how the host's resources are allocated among these groups [20].

3. Related Work

In this chapter, we refer the reader to existing works in the field of flaky tests and explain how they relate to this thesis. We categorize the literature into four areas. First, we cover literature describing the root causes of flaky tests. Then we provide a summary of existing detection strategies and their limitations. The third area addresses work that developed metrics to measure the severity of flakiness. In the fourth section, we refer to work focusing on fixing flaky tests. We end by emphasizing the research gap that this thesis aims to close.

3.1. Root Causes

Luo et al. [4] grouped flaky tests into ten root cause categories and found that the most common cause for flakiness apart from *test order dependency* is *concurrency*.

Romano et al. [5] studied flakiness with a focus on UI tests and extended the set of root causes with categories related to the environment and interactions with the test runner API. They confirm that concurrency-related flakiness makes up the majority of cases.

Eck et al. [1] interviewed professional developers about the nature and origin of the flaky tests they encounter. Apart from confirming the occurrence of seven categories by Luo et al., they also add four new categories of flakiness: *too restrictive range*, *test case timeout*, *platform dependency* and *test suite timeout*. Developers classified the last three as especially hard to fix.

3.2. Detection Strategies

Rerun remains a common strategy to detect flaky tests [21]. The main drawback of this approach is that it can take many runs to produce a flaky failure which is costly in terms of time and resources [2, 4].

DeFlaker by Bell et al. [22] is an alternative approach that, for each failed test, analyses whether it executes the code changed by the latest code revision. If not, the failure

cannot have happened due to regression and is marked as flaky. While this approach can reduce the detection effort significantly in some cases, it cannot detect flakiness in certain situations. If a developer tries to fix a flaky test by modifying its code, the test will definitely execute these changes, and no estimation about flakiness can be made.

IDFlakies by Lam et al. [15] starts by establishing an *original order* in which tests are run. They run the tests on this original order a set number of times and flag each test that shows inconsistent behavior as NOD flaky. They run the remaining tests in varying orders, and if a failure occurs, they rerun the test both in the original order and in the order that caused the failure. If the test again passes on the original order but fails on the order that caused the initial failure, the test is marked as OD flaky, otherwise as NOD flaky. The authors found that the distribution of tests between OD and NOD flaky tests is about equal, which shows that the challenging-to-detect NOD flaky tests make up a large portion of the total cases. This finding further increases the need for an efficient detection strategy which we aim to provide with our approach.

FlakeScanner by Dong et al. [23] focuses on concurrency-related flakiness by systematically exploring the space of all possible event orders in multi-thread scenarios. If a test passes for one order of events but fails for another, it is marked as flaky. For this purpose, they first trace events during regular execution and establish chronological relationships between them. They then calculate a space of possible execution orders and utilize Android's debug mode for scheduling events programmatically in these orders.

Shaker by Silva et al. [6] works by starting additional stressor tasks on the system executing the test. By adding noise to the execution environment, they aim to influence the order of scheduled events, which can change a test's output. While the authors report significant improvements in detection rates for known flaky tests, they do not discuss how resource contention influences previously stable tests.

3.3. Metrics for Severity

The extent to which a test exhibits flaky behavior can be measured with various metrics. While we will use the *flake rate* for our work, previous practitioners have introduced different metrics.

For example, Kowalczyk et al. treat the results of a test as a series of two unique values (pass and fail) and calculate the resulting entropy of the series. They also include a *flip rate*, which quantifies how often the results switch between the two values [24].

Machalica et al. choose a different approach and, given a history of results, provide a probability distribution for the flakiness score of the test. If the distribution is concentrated around a particular point, the corresponding flakiness score likely expresses the test's flakiness [18].

3.4. Fixing Flaky Tests

Some work focuses on predicting the underlying root cause of flaky failures. This prediction gives the developer a rough estimate of what kind of code could be responsible for the failure, speeding up the debugging process.

RootFinder by Lam et al. [12] instruments code to generate log files that include instances of method calls with return values and exceptions. Next, the tool evaluates a set of predicates that describe specific behavior for each instance of a method call. Then, the predicate evaluation for passing and failing test executions is compared and checked for differences. If a predicate consistently evaluates to one value (true or false) for passing executions and the opposite value for failing ones, it likely describes the type of flakiness well.

IFixFlakies by Shi et al. [25] looks for so-called *helpers*, which are tests in the suite that manipulate shared state in a way that makes OD flaky tests pass. The framework then suggests code changes utilizing these *helpers* to fix the flaky tests.

Other work examines the process of fixing flakiness from the developer's perspective. Eck et al. [1] interview professional developers about fixing strategies and challenges regarding flaky tests. The most common fixing strategy is adding a *waitFor* statement, while the biggest challenge developers report is reproducing the context that leads to the flaky failure.

3.5. Research Gap

Our approach aims to provide an efficient detection strategy for NOD flaky tests. Specifically, we want to target concurrency-related flakiness as it comprises the largest portion of flaky tests found in literature [1, 4, 12, 14]. For this purpose, we want to limit the available resources to influence the order of events between threads, increasing the likelihood of flaky failure. While other practitioners have presented similar strategies [6], our approach differs in three ways.

Firstly, while previous work found evidence that interfering with available resources can increase the flake rate of flaky tests, they did not study the effect on stable tests. As previously mentioned, while in practice, virtually all tests show flaky behavior to some extent, the severity with which they do varies significantly between tests [18]. Thus, especially in larger projects, it can be helpful to prioritize those flaky tests that are most likely to cause problems during development. Therefore, besides measuring how rapidly our approach can detect flaky tests, we also want to examine how many stable tests are flagged as flaky. These tests would not, or only rarely, fail under regular development circumstances.

Secondly, we are researching end-to-end tests that use the Cypress testing framework. Most of the related work in the field of flaky tests so far has dealt with unit tests. Given the increased complexity of end-to-end tests compared to unit tests, the room for unexpected and non-deterministic behavior increases, making them a fitting subject for research on flakiness [5, 26]. Furthermore, since the run time of end-to-end tests is usually longer than that of unit tests, the cost of rerunning them is higher. Thus, the reduction in runs required to detect flakiness that we aim to achieve with our approach is crucial. While some projects examined by previous work include end-to-end tests, none use Cypress. Given Cypress' growing popularity, research on flaky tests utilizing this framework could help mitigate the issues caused by flakiness in various projects.

Lastly, our approach limits the resources through Docker instead of putting real stress on the system. This approach may be more economical and manifest different flakiness due to the mechanism used to interfere with resource usage.

4. Approach

On a high level, we want to limit the resources of the system executing the tests in a way that influences the order of events for concurrent execution. We hypothesize that, as a result, flaky tests of the concurrency category will fail more often, allowing us to detect them with fewer runs. While higher limitations may increase the likelihood of detecting flaky tests, it may also increase the chance of making stable tests fail. Therefore, we start by classifying tests into flaky and stable by investigating historical test results. We then draw a connection between the flakiness caused by concurrency and computing resources. Once we identify relevant resources, we investigate how much we should limit each resource. We create a handful of sample configurations and execute the tests with each configuration multiple times. Each configuration receives a rating based on how many flaky failures it manifested and how many stable tests were falsely flagged. The best configuration is then used to detect flakiness in practice.

4.1. Classifying Tests into Flaky and Stable

In order to assess the performance of our limitation configurations, we need to classify the test suite into flaky and stable tests. For this purpose, we fetch all available test results from the CI system. For each test result, we know the name of the test case, whether it passed or failed and which code version it ran on.

We can gather even more results by utilizing Cypress' *retry* feature, which makes it possible to rerun a test up to a certain number of times in case of failure. As soon as one of the runs passes, the *retry* process stops, and the test is marked as passing. If all the runs fail, the test is marked as failed [27]. While this feature is helpful during development to prevent flaky tests from letting builds fail, it does not accurately capture the flakiness of tests. A test that fails two times through Cypress' *retry* feature before it passes is flaky but cannot be detected by inspecting the regular CI results since only the final result is recorded.

To make these additional runs explicit, we examine the log files of the CI platform and search for output from Cypress containing the keyword *'RERUN'*. A test must

have failed in the last run each time it is retried. Therefore, for each instance of the keyword, we extract the name of the affected test and create an explicit *failed* result that we append to the regular results.

Now that we have gathered all individual test results, we group them by the test name and code version. We filter to only include groups with at least one passing and one failing result and calculate the flake rate. To make results more statistically significant, we only include groups with at least five results. We now have a list of all flaky tests per code version with their respective flake rate. We focused on a code version with many available results per test and a high average flake rate. We will elaborate on the details in chapter 5.2.

4.2. Connecting Computing Resources and Flakiness

Now that we know which tests are flaky and which are stable, we want to introduce resource limitations to increase the flake rate of flaky tests. For this, we first explain how concurrency-related flakiness relates to computing resources.

Concurrency can lead to the non-deterministic execution of events between threads. Some of the possible orders of events may lead to undesirable behavior, such as race conditions and deadlocks [4]. While non-determinism can be the correct behavior of the SUT, if a test does not account for all the possible orders of execution, it may pass for some orders while failing for others, resulting in flaky behavior. Some of these orders of events may occur less often during regular execution. As a consequence, the test passes most times while failing rarely, making it difficult to detect.

If we limit the CPU, the resulting resource contention leads to the operating system scheduling processes and their threads differently depending on their priority. Moreover, more context switches may be necessary to prevent the starvation of processes. These switches add overhead that slows down the system, which can further influence execution order. By exploring orders different from regular test execution, we aim to increase the likelihood of flaky failures. Let us consider an example:

An online learning platform lets students submit coding exercises and receive automatic feedback. One part of the application asks the user to enter the code for a sorting algorithm into a text field. The user can then click on a button to submit the code, and after receiving a reply from the server, feedback for the submission gets displayed. A test prone to flakiness might be implemented as follows.

The test runner enters the appropriate code for the requested algorithm into the text field and then submits the exercise via the button. After waiting for the absolute time of five seconds, the test checks the page for a string containing the expected feedback. The test passes or fails depending on whether the correct feedback gets displayed.

In most cases, five seconds will suffice for the server to execute the student's code, compare the result and respond with feedback. However, if the server's CPU is limited, it may take longer to execute the code, the response may not be sent within five seconds, and the test runner does not find the expected feedback string on the page, resulting in a flaky failure. The issue can be solved by adding an explicit *waitFor* statement that properly waits for the server's reply before checking for the feedback string.

Another computer resource is the main memory (RAM). If a process reaches its maximum amount of RAM and requests more, the operating system may resort to paging out the data to disk storage. This pagination leads to a significant performance penalty which can influence the ordering of events [28].

Developers at Google discovered a correlation between the RAM used by a test and its severity of flakiness. The more RAM a test uses, the higher the probability of it being flaky [9]. The reason for this correlation could be manifold. For example, tests that require more RAM may be larger and more complex, thus leaving more room for non-deterministic behavior. However, it could also be that these tests are more likely to be influenced by spikes in workload on the testing infrastructure. As the available RAM gets less, pagination increases and possible timing issues become apparent. Therefore, limiting the system's RAM may increase the likelihood of flaky failures.

4.3. Running Tests with Varying Resource Limitations

Now that we know which resource type we should limit, namely CPU and RAM, we must decide on the right degree of limitation. Intuitively, more severe limitations may result in more significant differences in the order of events, increasing the chance of a flaky failure. However, if the limitation is too severe, previously stable tests could fail, or the entire test suite could crash due to an out-of-memory exception [29]. Therefore, we need to determine a reasonable degree of limitation that manifests concurrency-related flakiness while not interfering with stable tests or with the test suite as a whole.

We create a set of limitation configurations, each limiting a specific resource to varying degrees. We also create mixed configurations that limit multiple resources at the same time. Since we study end-to-end tests with Cypress, we have not one but three processes involved with testing: a browser process, which executes the front-end code, a Node.js

server process used by Cypress to communicate with the browser and send commands, as well as the server process. Our testing setup uses containerization with Docker, and both the browser and Node.js server processes run in the same container. Therefore, we will summarize these two processes as *client* and treat them as a single entity.

With two resources to limit and two processes (client and server), this results in four parameters that we can manipulate in our configurations: CPU_{client}, CPU_{server}, RAM_{client} and RAM_{server}. In order to generate promising configurations, we use an approach that is similar to a binary search. Let us look at the limitation configurations that limit only one resource at a time.

We start with an upper and a lower bound for the parameter. We will discuss how we choose the initial values shortly. The upper bound represents a configuration that yields promising results, which in this context means that flaky tests fail while stable tests pass. As we could not predict how different configurations would behave, we manually decided whether a configuration was promising on a case-to-case basis.
The lower bound is a configuration for which the limitation is too severe, and the test suite as a whole starts to fail. We then create a new configuration that considers the midpoint between the upper and lower bound as its limitation parameter. If this parameter yields promising results, it becomes the new upper bound. If not, the limitation is assumed too severe and becomes the new lower bound. While this procedure can be repeated indefinitely to generate more configurations, we limit ourselves to three iterations.

We derive the initial upper bound through the analysis of historical resource usage. We track the resource use of both the client and the server process during regular test executions and measure the 90% quantile of each parameter. That is the value for which 90% of all the observed usages were smaller or equal. One could also choose a more restrictive limit or the average value. However, we opt for a more inclusive approach since we cannot predict how severely even minor limitations will influence the tests' behavior. The initial lower bound is a dummy bound of zero for all resources. In chapter 5.4, we will list the resulting configurations, including those that simultaneously limit multiple resources.

We utilize Docker's runtime options to implement the resource limitation. Since we want to limit the CPU and RAM of the processes, two options are of interest: *cpu quota* and *mem limit* [19].

The *cpu quota* option works in conjunction with *cpu period*. Together, the two provide a mechanism to control the CPU time a process can use during a given time interval. The period governs the time slice within which processes are assigned a certain quota

of that slice. If a process has consumed all its quota and requests more, it is throttled until the next period begins.

For example, if we set the quota to 50000 and the period to 100000 (default value), the respective process will have access to half of the time on one core. In a multi-core system, the entire available period is the number of cores times the specified period. If we set the quota to 200000 and keep the default period, the process will be throttled to using two cores [30]. It should be noted that the CPU usage in percentage is measured relative to one core, meaning that usages above 100% are possible. For example, 150% CPU usage refers to using the complete compute time of one core and half of the time of a second core.

For the *mem limit* option, we provide the maximum amount of RAM a process should be able to use. If the process needs to allocate more memory, the operating system resorts to swapping the data to disk, which slows down the process.

In our setup, the containers for the client and server test processes are deployed through Docker Compose, a tool that enables managing multi-container Docker applications. The user specifies services, their dependencies and configuration parameters in a file and can start and manage them efficiently through Compose. To limit the resources, we append an entry with the *cpu quota* or *mem limit* option to the respective service inside the configuration file. We then commit these changes to a new branch in the git repository of the project.

The CI system connected to the repository registers the change and automatically runs the tests once with the new configuration. We then trigger the remaining runs via a script that interacts with the API of the CI system. Since we use the development pipeline, we run our tests at night to not slow down developers' tests. Furthermore, we perform our tests over multiple days to try and filter out noise from failures due to environmental issues like crashing external services, downtimes or connection issues.

Performing the tests in the development pipeline instead of a local environment has the advantage of being as close as possible to the workflow in practice. Flakiness is often difficult to reproduce due to the numerous environmental factors that can influence the test execution, such as the operating system used or the local version of the testing framework. For example, Lam et al. found that 86% of the tests they studied were only flaky in the CI pipeline and not on local machines [12].

4.4. Measuring Configuration Performance

We now have results for various limitation configurations and want to find the best-performing one. For this purpose, we use metrics from the field of binary classification. We assume that each test has a true label of either *flaky* or *stable*. We attain these true labels from historical test results as explained in chapter 4.1. We then treat each test result obtained through our configurations as a prediction of whether the test is flaky. The predicted label is *stable* if the test passes. If it fails, the predicted label is *flaky*. If the test suite as a whole fails, we treat every test as failed.

As previously mentioned, we not only want to consider how many flaky tests are correctly labeled as such (true positives) but also how many stable tests are falsely flagged as flaky (false positives). We will elaborate on the metrics used in chapter 5.3.1.

The best configuration is then used to detect flakiness. If our approach works as intended, flaky tests will frequently fail, which may lead to only failures being observed for a test. Both passing and failing results are needed to mark a test as flaky. Otherwise, the failures may indicate an actual bug introduced by regression. Therefore, performing additional regular runs without limitation may be necessary to generate passing results. However, we predict the overhead for this to be negligible as only a small amount of runs are needed to confirm whether a failure is flaky. For example, Gruber et al. found that only one rerun of a failed test is needed to detect flakiness with a statistical confidence of 95% [2]. Moreover, passing results may already be available from the regular CI workflow.

5. Evaluation

In order to evaluate our approach, we first formally define the research questions (RQs). We then take a closer look at our study subject and its tests. After explaining the setup of our evaluation, we present the results. Lastly, we discuss our findings and possible future work in the field of flaky test detection.

5.1. Research Questions

We want to answer whether limiting resources is a more efficient detection strategy than regular *Rerun*. An efficient approach is both correct, meaning that the right tests are detected as flaky, and it minimizes the effort for detection. We derive two research questions:

RQ1: How accurately do resource-limited runs detect flakiness?

RQ2: How much detection effort can we save by using resource-limited runs?

A common metric for the effort of detection is the number of runs necessary to detect flaky behavior [2]. However, since we expect the resource-limited runs to take longer to execute, we also measure the absolute time it takes to detect flakiness.

We derive the following sub-questions for RQ2:

- **RQ2.1**: How many runs can we save using resource-limited runs?
- **RQ2.2**: How much absolute time can we save by using resource-limited runs?

5.2. Study Subject

The study subject for our experiments is the Artemis project of the Chair of Applied Software Engineering at the Technical University of Munich. Artemis is a web application that serves as an interactive learning platform that allows students to complete various exercises and receive feedback on their submissions. The server application is built with Java and leverages the SpringBoot framework, while the client is an Angular application written in TypeScript. The Artemis team uses the Cypress framework to write end-to-end tests for their application. They use Atlassian Bamboo as the CI platform that builds the application, deploys it in a testing environment and then executes the test suite.

We retrieved build results between December 2021 and February 2023, which amounts to 3262 builds, excluding those with no test results due to a failure during the build process. In total, we obtain 189285 individual test results.

Since early 2023, the Artemis team enabled Cypress' *retry* feature, which automatically reruns failed tests to check if the failures were flaky. If the test passes during any of the runs, it is marked as passing. In Artemis's case, the system retries a test at most twice. We would miss these implicit retries if we only considered the final test result. Therefore, we scan the log files of Bamboo to find out which tests were retried and add an explicit failed test result for each retry. Through this process, we gain an additional 4006 results.

The test suite grew from one test at the beginning to 84 tests at the end of the observation period. During that time, 47 tests showed flaky behavior distributed over various code versions. The total flake rate is 0.87%. The proportion of flaky failures to total failures is 16.96%. We find that if a failure happens, the probability of it being flaky is significant. Furthermore, a large portion of the test suite showed flaky behavior at some point during the observation period.

Our experiments focused on the code version at commit *49e1f2b4ac*. Apart from minor changes for flakiness detection, this code version is the same as the one at commit *2258602cb1*, created in September 2022. At this point, the test suite consisted of 58 tests. The median execution time of the test suite was 31 minutes and 42 seconds. The reason for focusing on this particular version is that a comparatively high amount of data points are available. The average number of results per test is 21, and the average flake rate is 12.69%. While Bamboo usually only runs the tests once per commit, developers can run additional, manual runs.

5.3. Setup for Evaluation

5.3.1. Evaluation Metrics

As described in chapter 4.4, we treat the outputs of each configuration as predictions in a binary classification problem. Each test in the suite has a true label of *flaky* or *stable*. The test results of each configuration are then the predictions of these labels. Speaking in terms of a *confusion matrix*, *flaky* translates to *positive* and *stable* to *negative*. We then apply common metrics for binary classification to evaluate the configurations' performance.

Recall

The recall measures the portion of all positives correctly identified as such. In the context of our scenario, it allows us to estimate how many of the flaky tests our approach detects. It is calculated as:

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{5.1}$$

Precision

The precision measures what portion of predicted positives are true positives. It allows us to estimate how many tests flagged as flaky are stable and thus false positives. It is calculated as:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{5.2}$$

F1 score

We consider both precision and recall to evaluate a configuration's overall performance. Therefore we measure the F1 score as follows:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{5.3}$$

Metrics for Detection Effort

In order to statistically estimate how many runs are necessary to detect flakiness, we introduce some metrics in the following [2]. As mentioned in chapter 4.4, we assume that for each test not broken by regression, a passing result either already exists or can be generated with little effort. We only need to generate flaky failures. Therefore, the probability of detecting flaky behavior for the test t, after n runs, is calculated by taking the complement of the probability to exclusively observing passes:

$$U_t(n) = 1 - P_t(PASS)^n \tag{5.4}$$

Given $U_t(n)$, we can calculate the minimum number of runs needed to manifest flakiness with probability p:

$$n_{t,p} = \min\left(\{n | U_t(n) > p\}\right) \tag{5.5}$$

The number of flaky tests detected after n runs with confidence x is defined as:

$$S(n, x, T) = |\{t \mid n_{t,x} \leq n \wedge t \in T\}| \tag{5.6}$$

5.3.2. Evaluation Baselines

For classification, we choose two different baselines. The first is a dummy classifier that marks all tests as flaky. We will use the average over results from the least-restrictive limitation for each resource for the second baseline.

For effort reduction, we will treat historical test results as *Rerun* and use it as our baseline. We will use the fail rate to calculate how many runs it would take to detect a certain amount of flakiness, and we will use the median build time to decide which approach performs better in terms of absolute execution time.

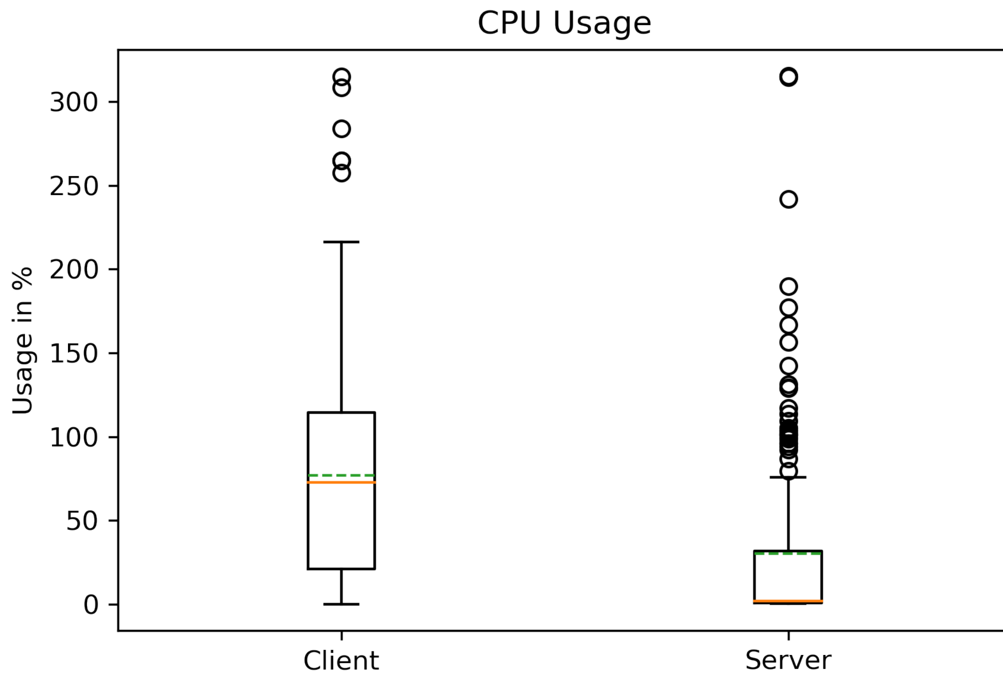

Figure 5.1.: Distribution of CPU usage for server and client during regular test execution

5.4. Results

In total, we created 18 different configurations. We started by looking at the resource usage during regular test execution by sampling the resource use of the client and server processes 200 times during regular test execution. The build agents executing the tests have a total capacity of six logical cores and 12GB of main memory.

Figure 5.1 depicts the distributions for the CPU usage. Looking at the distribution, it becomes apparent that, on average, the client requires more CPU than the server. The client shows a greater spread in usage than the server. The server does, however, show a few outliers with high usage. Both the client and the server show a similar maximum usage.

Figure 5.2 depicts the distributions for the RAM usage. The server generally consumes more main memory than the client. Again, the client shows a greater spread in usage while the server uses a more consistent amount. Both the client and the server show some outliers.

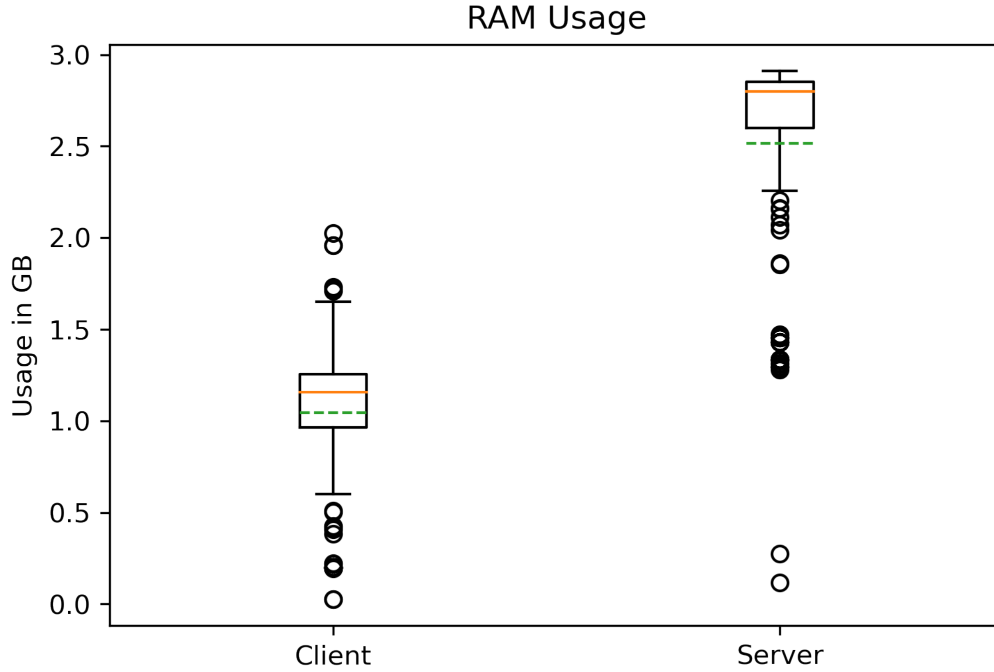

Figure 5.2.: Distribution of RAM usage for server and client during regular test execution

Prior to the conduction of the experiments, it was unknown how severely even minor resource shortages would affect the fail rate of the tests. Therefore, as an initial upper bound for the limitation, we chose roughly the 90%-quantile of each resource. This approach reduces the range of analysis and still avoids overly constraining limitations. The initial dummy lower bound is zero or no resources. The resulting limitation parameters are listed in table 5.1. The two mixed configurations simultaneously use each resource's first and second limitation parameters. We ran each configuration ten times.

CPU_{client} in %	150	75	56.25	37.5
CPU_{server} in %	100	50	37.5	25
RAM_{client} in GB	1.5	1.125	0.75	0.375
RAM_{server} in GB	3	1.5	1.125	0.75

Table 5.1.: Maximum usage parameters used in the limitation configurations

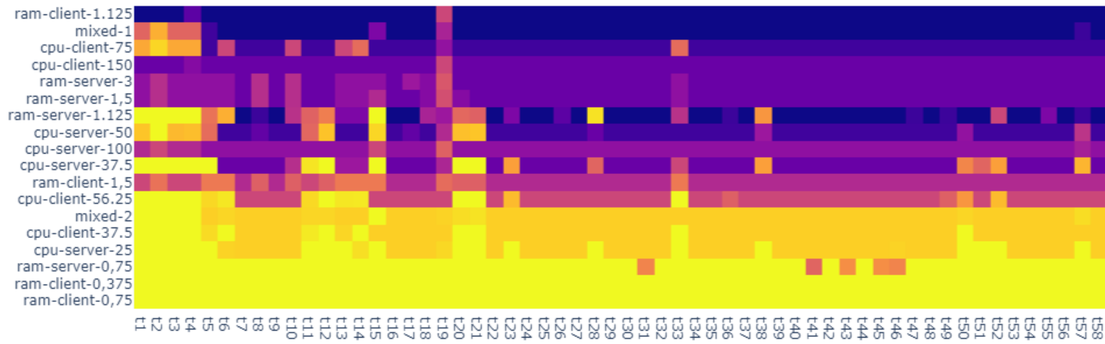

Figure 5.3.: Fail rates per test per configuration - brighter colors indicate higher fail rates

Figure 5.3 depicts the fail rate of every test for the individual configurations. Generally, more restrictive limitations lead to a higher failure rate. Certain tests, such as *t2* and *t19*, react early, showing increased fail rates even at less restrictive limitations. We expect these tests to be more prone to flakiness caused by concurrency issues. On the other hand, the tests *t39* to *t46* are examples of tests that only react at relatively strong limitations.

When a specific limitation is reached, virtually all tests show a high failure rate. These configurations are of little help to the developer as the implication would be that the entire test suite has to be checked for flakiness, rendering a detection strategy pointless. This finding shows that when optimizing a flaky detection strategy, it is important to track the fail rate of stable tests instead of only trying to increase the fail rate of known flaky tests, like in the approach by Silva et al. [6].

One interesting question is whether limiting certain resources has a more significant effect than others. The average fail rates per resource are depicted in figure 5.4. While there is a slight difference, the resources CPU_{client}, CPU_{server}, and RAM_{server} generally show a similar trend in failure rate. RAM_{client} shows surprising behavior since the failure rate of the configuration 1.5GB is higher than the one of 1.125GB, even though it should be less restrictive. Upon closer inspection, we suspect that the increased failures with 1.5GB are due to infrastructure problems, as most of the failures come from cases where the test suite as a whole failed to execute. Moreover, the fail rate steeply increases between 1.125GB and 0.75GB. An implication may be that the client is more sensitive to RAM limitations. This would be surprising as the RAM usage of the server is generally higher with more outliers, as depicted in figure 5.2. Overall, no clear statement can be made about whether the CPU or the RAM limitation has a more substantial impact when only considering the failure rate.

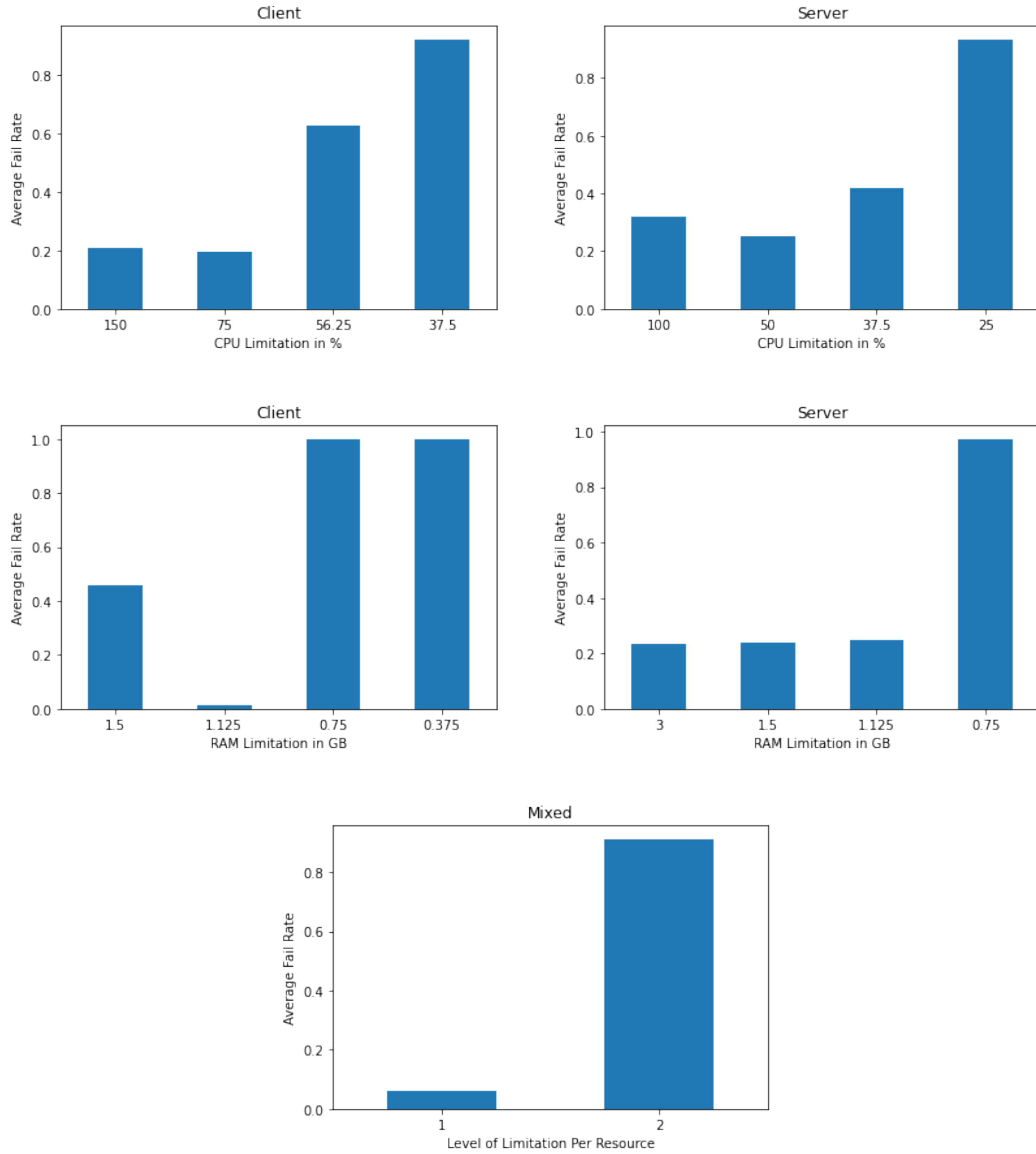

Figure 5.4.: Average fail rates per resource

Looking at the results for the mixed configuration, a synergy between the limitation of different resources becomes apparent. Level one applies the least restrictive limitation of each resource simultaneously, and level two applies the second-to-least one. While both levels show a low failure rate when applied independently, the second mixed level shows a sharp increase in failure rate.

5.4.1. Accuracy of Detection

Next, we will examine how accurately the configurations detected flaky tests while minimizing false positives. Of the 58 tests in the suite, 11 were marked as true label flaky. A list with the method names of all flaky tests can be found in appendix A.1.

Table 5.2 shows performance metrics for the two baselines and the five best-performing configurations. Appendix A.1 contains the performance for all remaining configurations.

The best configurations are all related to the CPU resource. While both client and server-related configurations are included, the top two configurations are related to the server. This result is interesting since the average CPU usage of the client is higher than that of the server - as depicted in figure 5.1. One possible explanation could be that much of the concurrency-related flakiness is caused by timeouts on the client side. If we slow down the server, the likelihood of a flaky failure due to client-side timeout increases.

All five configurations perform better than the baselines. The best configuration is the one that limits the server's CPU to 37.5%. It achieves an F1 score of 0.53, which is about 65% and 89% better than baseline one and two.

	Precision	Recall	F1 Score
dummy-all-positive	0.19	1	0.32
dummy-least-average	0.28	0.34	0.28
cpu-server-37.5	0.4	0.77	**0.53**
cpu-server-50	0.43	0.6	0.5
cpu-client-56.25	0.34	0.88	0.49
cpu-client-37.5	0.27	0.97	0.43
cpu-server-25	0.25	0.97	0.4

Table 5.2.: Performance of the top five configurations and baselines

The correct limitation amount is important as the best score is reached for a configuration that is neither the least nor most restrictive.

While we achieve relatively high values for the recall, the precision is below 50% for all configurations, which means that a significant number of stable tests are flagged as flaky. In chapter 5.5.2, we will elaborate on this finding.

> **Answer to RQ1:** Our approach performs better than both dummy baselines. We achieve an F1 score that is 65% higher than baseline *dummy-all-positive* and 89% higher than baseline *dummy-least-average*. CPU limitation yields the most promising results, especially when applied to the server process. While the performance of our approach exceeds the baseline's, the precision is relatively low, with no configuration exceeding 50%.

5.4.2. Reduction of Detection Effort

We will examine the number of runs needed to detect flakiness for our approach and regular *Rerun*. Since our approach limits the number of resources, which slows execution, we will also look at how both approaches compare when looking at the absolute time it takes to detect flakiness.

The left graph of figure 5.5 shows that our approach requires significantly fewer runs to detect flakiness. Of the 58 tests in the test suite, 11 showed flaky behavior. Our approach needs only 14 runs to detect all flaky tests with a confidence of 95%, while regular *Rerun* requires 59. The resulting speedup amounts to almost 80%.

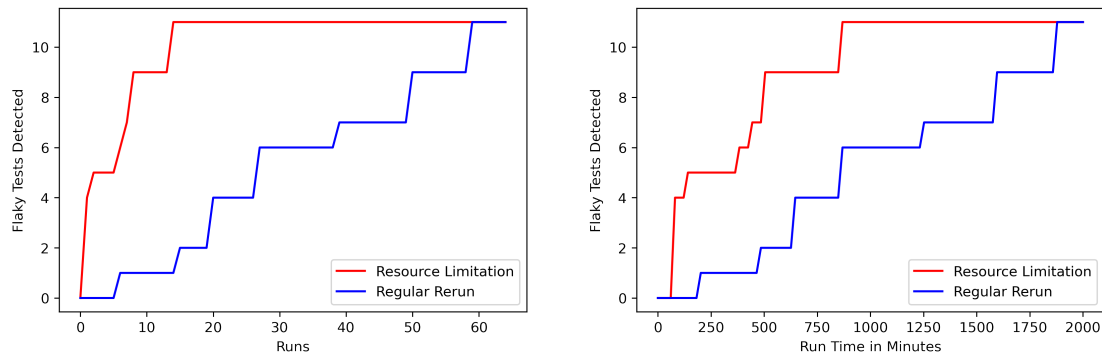

Figure 5.5.: Effort of detection for regular and resource-limited rerun $S(n, 0.95, T)$

Answer to RQ2.1: Our approach requires significantly fewer runs than regular *Rerun*. We can reduce the runs necessary to detect all flaky tests with a confidence of 95% from 58 to 14, which amounts to a speedup of almost 80%.

Looking at the right graph of figure 5.5, we can see that even though the difference is smaller, our approach outperforms *Rerun*. The median execution time of the test suite for *Rerun* is 1904 seconds, and for our approach, 3721 seconds. To detect all flaky tests with a confidence of 95%, we require around 1870 minutes with *Rerun* and 870 minutes with our approach, which translates to a speedup of around 50%.

Answer to RQ2.2: Our approach performs significantly better when looking at absolute run time. We can reduce the time necessary to detect all flaky tests with a confidence of 95% from 1870 to 870 minutes, translating to a speedup of around 50%.

To make our results comparable with existing literature, we follow the evaluation by Gruber et al. and answer the following three questions for both our approach and the baseline [2]:

(a) "How many times do I have to rerun a test to be 95% sure that the test is not flaky?"

(b) "If I rerun my tests ten times, which portion of the existing flakiness can I expect to find?"

(c) "How many reruns do I need in order to find 80% of all existing flakiness?"

The results are depicted in table 5.3. Our approach performs better on all three of the questions. It should be noted that the amount of runs necessary to detect flakiness is significantly smaller than the values obtained by Gruber et al. for Python tests. However, the time needed to execute Cypress end-to-end tests tends to be much higher.

Answer to RQ2: Our approach can detect flaky tests significantly faster than *Rerun*. When looking at the number of runs, we can achieve a speedup of almost 80%. While the performance difference is smaller for the absolute run time, our approach still outperforms *Rerun* and achieves a speedup of about 50%.

	Rerun	Our Approach
a) "How many runs to classify a given test?"	27	6
b) "Number of detected tests with ten runs?"	9.1%	81.8%
c) "How many runs to find 80% of flaky tests?"	50	8

Table 5.3.: Comparing detection effort for *Rerun* and our approach in terms of number of runs required for detection

5.5. Discussion

5.5.1. Assessing the Reproducibility of Flaky Failures

While these results seem promising, the question arises whether we are reproducing the same flaky failures as they happened in the past or if the tests now fail due to different reasons, which could be interesting from a debugging perspective. To investigate this, we looked into the most common error messages, both during regular execution in the past and executions with our approach.

One issue we encountered was that we could not retrieve error messages for failures retried automatically by Cypress. The reason for this is that, for the retry feature, we only have access to the logs produced by the Cypress runner, which do not include error messages. As a result, no clear reason or error message can be derived for a significant portion of failures, both for regular runs and our approach.

Another issue is that of *testless builds*. During our experiments, sometimes the test suite as a whole failed, in which case no test results could be retrieved. For our classification, we decided to mark all tests as failed in such cases.

For the cases where we can retrieve meaningful error messages, the results seem to be more or less equal for both regular runs and our approach. Either the error messages are the same or include small deviations. For example the test *Instructor can see complaint and reject it*, two similar error messages are:

1. *AssertionError: Timed out retrying after 20000ms: Expected to find element: '#response-TextArea', but never found it.*
2. *AssertionError: Timed out retrying after 20000ms: Expected to find element: '#show-complaint', but never found it.*

In conclusion, we cannot clearly state how well our approach reproduces specific reasons for failure. Measures to achieve more meaningful results in this area would be to somehow get access to the error messages of failures retried by Cypress or to treat testless builds differently, for example, by introducing a *skipped* category. The latter would require an adaption of our present binary classification approach. It should be noted that a flaky test can produce different kinds of undesired behavior, which could lead to varying error messages.

5.5.2. Analysing the Low Precision

Besides our approach's high recall, we noticed relatively low values for precision, which suggests that many stable tests started to fail as we introduced resource limitations. Since we aim to detect the flaky tests that cause problems during development, these false positives are undesirable as they lead to efforts being spent on fixing flakiness which may only rarely occur in practice. As a consequence, fixing more severely flaky tests is delayed. If the false positive counts are indeed as high as they appear, it would be a drawback of our approach. We investigated the reasons for these failures and identified two issues that may distort the number of false positives: testless builds and the set of tests we consider flaky not being inclusive enough.

A testless build is a build for which the execution of the test suite as a whole failed. For our binary classification, we treat such a case as if every test in the suite failed. Two out of the ten runs for our experiment were testless builds. That means each stable test has a fail rate of at least 20%. One possible solution would be to ignore testless builds in the performance evaluation of the configurations. However, configurations that repeatedly produce testless builds are most likely undesirable as they lead to wasted computing resources in the CI. Therefore, testless builds would need to be punished in some other way during the performance evaluation of the configurations.

Another reason that leads to an increased number of false positives is the way we classify flaky and stable tests. For our approach, we only marked a test as flaky if it showed passing and failing runs on the code version of the commit *49e1f2b4ac*. Some tests that produced apparent false positives *do* show flaky behavior on many other code versions, just not on the version we used for our experiments. Therefore, failures produced by these tests may actually be true positives.

One way to approach this issue could be to consider more than a single code version to decide whether a test is flaky or stable. In that case, one could choose a time frame and consider only commits made within to minimize the possibility of falsely flagging a real failure due to regression as flaky. If we choose this more inclusive

	Precision	Recall	F1 Score
dummy-all-positive	0.53	1	0.7
dummy-least-average	0.59	0.3	0.38
cpu-client-56.25	0.64	0.81	**0.72**
cpu-client-37.5	0.57	0.95	0.71
cpu-server-25	0.55	0.96	0.7
mixed-2	0.55	0.93	0.69
cpu-server-37.5	0.68	0.67	0.67

Table 5.4.: Performance of the top five configurations and baselines for a more inclusive classification of flaky tests

classification, 31 of the 58 tests show flaky behavior, and our approach performs better. The improved performance is depicted in table 5.4. We see that now the best configuration is *cpu-client-56.25* with an F1 score of 0.72. We also see that the difference to baseline *dummy-all-positive* is smaller, which is to be expected, given the high amount of flaky tests. The extended list of all 31 flaky tests can be found in appendix A.2.

However, we also found some tests that match our initial definition of false positives. These tests never showed flaky behavior during regular development but started doing so when a specific amount of limitation was reached. When looking at the distribution of failures, we find that about half are true positives. Roughly a quarter are false positives which would be true positives with a more inclusive classification of flaky tests. The remaining quarter of failures is comprised of false positives that never showed flaky behavior in the past and thus would be false positives even with a more inclusive classification.

In conclusion, while our approach produces false positives, the actual number may be distorted due to how we handle crashing CI builds and categorize tests into flaky and stable before the experiments. While the concept of stable tests failing under limitation does show in practice, the proportion these cases make up compared to all failures may be smaller than expected.

5.5.3. Future Work

For this study, we had to limit ourselves to Cypress end-to-end tests for the Artemis project. Since flakiness affects many projects with different kinds of tech stacks, our approach could be tested on a different data set to assess the consistency of our findings.

While our study focused on CPU and main memory, other resources like network bandwidth could also be considered in the limitation configurations.

We used Docker to limit the number of resources that processes can use. Previous studies used real stressor tasks that compete for the resources instead [6]. Future studies could look into the difference between these two approaches. Possible differences could include which tests can be detected as flaky, how rapid detection works or whether limiting resources instead of competing for them can reduce the cost of the testing infrastructure.

Other practitioners may also look into the issues we presented in the discussion. For example, one could examine in detail how well detection approaches for flakiness recreate the same failures that occurred during development. Replicating the same kind of failure could be valuable for debugging and fixing the flakiness. There is also the issue of treating testless builds and false positives.

Alternatively, one could choose a different technique for limitation configurations altogether. For example, one could run the test suite on varyingly strong limitations and assign weights to the failures depending on the limitation applied. Failures during less restrictive limitations may be weighted more strongly than those that only occurred during very restrictive limitations. This approach would be similar to ours in that it focuses on failures most likely to happen during regular development.

6. Threats to Validity

6.1. Threats to External Validity

Since we focused on the Artemis project in our studies, we can only make claims about the effectiveness of our approach on tests that use a similar setup. Specifically, the tests we studied are end-to-end tests written in JavaScript using the Cypress testing framework. Therefore, our findings are not directly applicable to other programming languages or testing frameworks.

Since we used the development CI pipeline, we had to limit our experiments to ten runs per configuration to not interfere with tests by developers. While we could detect all of the flaky tests, more runs might help mitigate noise in the results, for example, due to infrastructure-related problems.

We use Docker to limit the resources of the testing process. Projects that do not use Docker for deployment can thus not directly apply our approach. Either the deployment method needs to be changed to Docker, or the resources could be limited through the cgroups feature of the Linux kernel, which is what Docker uses under the hood.

We use statistical methods to evaluate our approach to increase our external validity. This kind of evaluation includes all test results and allows for an objective comparison with other approaches [2].

6.2. Threats to Internal Validity

Throughout our research, the Artemis development team repeatedly updated the CI system and changed the workflow to execute the Cypress tests. In theory, the updates to the CI system change the environment in which the tests run, like a new version of Docker Compose. The changes to the testing workflow required us to apply some changes to the codebase for the tests to run properly. An example of such a change is the script for starting the Docker containers for the Cypress tests. At first, this script was configured directly inside Bamboo's build plan configuration. At some point, the

developers added the script as a shell file to the codebase and configured Bamboo to execute this file at the build's start. While we had to change the codebase to include this file, the behavior of the tests should not be influenced by this. These changes are unavoidable when working with an actively developed application and its testing infrastructure. To increase internal validity, the necessary changes to the code were reviewed by two practitioners independently and deemed as unlikely to influence the behavior of tests significantly.

Some of the runs we used to classify tests into flaky and stable came not from the regular development workflow but were manually triggered by another practitioner. These tests used the same code and were executed in the same environment as regular executions. The time of day at which execution happened and the interval between consecutive executions might differ from regular development, though, which could influence testing behavior unexpectedly. We deemed the risk for such unexpected behavior low and decided to take advantage of these additional runs to gather more data for our classification.

The fact that we focused on only the Artemis project for our studies increases internal validity as fewer factors had to be controlled to create reliable and reproducible results. Specifically, using the same CI infrastructure that is used in regular development helps to run the tests in a realistic environment. Furthermore, due to the relatively small amount of tests, we were able to manually analyze the results - as shown in the discussion section. We also included sanity checks by sampling the actual resource usage of processes during testing to validate whether our approach works as intended.

7. Conclusion

In automated regression testing, each change to the codebase triggers the execution of tests. If a test passes on the last version but fails on the current version of the code, it is assumed that the latest revision introduced a bug. However, this is not always the case in practice. Flaky tests produce both passing and failing results even though they are executed multiple times under the same conditions. This behavior undermines the purpose of automated regression testing and introduces numerous issues, such as delayed deployment processes or diminishing trust in the test suite by developers.

In order to fix flaky tests and mitigate these issues, a variety of strategies for the detection of flaky tests have been developed. *Rerun* is a common practice that executes each test repeatedly for a set amount of times under the same conditions. Since some flaky tests only fail rarely, this approach can require many executions to detect a test as flaky.

In this thesis, we presented a novel detection approach that focuses on concurrency-related flakiness. By limiting the available resources, we influence the order of events between threads, making flaky failures more likely. We focus on detecting those flaky tests that are most likely to cause problems during development by optimizing our limitation configuration for a high fail rate on known flaky tests while keeping the fail rate low on stable tests.

Our approach detected all flaky tests and achieved a recall of 77%, which is significantly higher than the 34% for one of the baselines. Moreover, by increasing the flake rate, our approach can detect flakiness more rapidly. We reduced the number of runs needed to detect all flaky tests from 58 to 14, translating to a speedup of almost 80%. When considering the absolute time it takes to detect flakiness, the difference is smaller, but we can still outperform *Rerun* by 50%.

While our approach yields promising results for detecting flaky tests, some previously stable tests also showed flaky behavior when executed with our configuration, leading to a relatively low precision of 40%. Through manual inspection, we found that in many of these cases, the affected test *did* show flaky behavior on many occasions in the past but simply not on the code version we chose to classify tests into flaky and stable.

If we choose a more inclusive identification, our approach performs significantly better with a precision of 64%, but this introduces the risk of falsely flagging failures due to regression as flaky. Therefore, additional research may be needed on properly handling false positive cases.

While this thesis focused on Cypress end-to-end tests for the Artemis project, experiments on other projects and tests should be conducted to further validate our approach. Apart from that, other resources could be considered for limitation, such as network bandwidth. In conclusion, limiting resources presents a promising strategy for reducing flaky test detection efforts. This thesis provides an initial approach and discusses possible improvements.

A. Appendix

A.1. Artemis Flaky Tests Strict

- Exam assessment Exam programming exercise assessment Assess a programming exercise submission (MANUAL)
- Programming Exercise Management Programming exercise creation Creates a new programming exercise
- Programming Exercise Management Programming exercise deletion Deletes an existing programming exercise
- Modeling Exercise Assessment Spec Handling complaints Instructor can see complaint and reject it
- Programming exercise participations Makes a partially successful submission
- Programming exercise participations Makes a successful submission
- Exam management Registers the course students for the exam
- Exam date verification Exam timing Student can start after start Date
- Modeling Exercise Participation Spec Student can start and submit their model
- Modeling Exercise Assessment Spec Handling complaints Student can view the assessment and complain
- Modeling Exercise Assessment Spec Tutor can assess a submission

A.2. Artemis Flaky Tests Inclusive

- Course management Manual student selection Adds a student manually to the course
- Exam assessment Exam exercise assessment Modeling exercise assessment Assess a modeling exercise submission

- Exam assessment Exam programming exercise assessment Assess a programming exercise submission (MANUAL)
- Exam assessment Exam exercise assessment Text exercise assessment Assess a text exercise submission
- Quiz Exercise Assessment MC Quiz assessment Assesses a mc quiz submission automatically
- Exam assessment Assess a quiz exercise submission Assesses quiz automatically
- Programming exercise assessment Assesses the programming exercise submission and verifies it
- Static code analysis tests Configures SCA grading and makes a successful submission with SCA errors
- Quiz Exercise Management Quiz Exercise Creation Creates a Quiz with Multiple Choice
- Quiz Exercise Management Quiz Exercise Creation Creates a Quiz with Short Answer
- Course management Course creation Creates a new course
- Programming Exercise Management Programming exercise creation Creates a new programming exercise
- Text exercise participation Creates a text exercise in the UI
- Exam creation/deletion Creates an exam
- Quiz Exercise Management Quiz Exercise deletion Deletes a Quiz Exercise
- Programming Exercise Management Programming exercise deletion Deletes an existing programming exercise
- Exam date verification Exam timing Exam ends after end time
- Exam management Manage Students Generates student exams
- Modeling Exercise Assessment Spec Handling complaints Instructor can see complaint and reject it
- Programming exercise participations Makes a failing submission
- Programming exercise participations Makes a partially successful submission
- Programming exercise participations Makes a successful submission

- Exam participation Participates as a student in a registered exam
- Exam management Manage Students Registers the course students for the exam
- Course management Manual student selection Removes a student manually from the course
- Exam date verification Exam timing Shows after visible date
- Quiz Exercise Participation Quiz exercise participation Student can participate in MC quiz
- Exam date verification Exam timing Student can start after start Date
- Modeling Exercise Participation Spec Student can start and submit their model
- Modeling Exercise Assessment Spec Handling complaints Student can view the assessment and complain
- Modeling Exercise Assessment Spec Tutor can assess a submission

A.3. Performance of Configurations

	Precision	Recall	F1 Score
cpu-server-100	0.22	0.37	0.27
cpu-server-50	0.43	0.6	0.5
cpu-server-37.5	0.4	0.77	0.53
cpu-server-25	0.25	0.97	0.4
mixed-1	0.44	0.19	0.27
mixed-2	0.22	0.93	0.36
ram-client-1.5	0.26	0.58	0.36
ram-client-1.125	0.45	0.08	0.14
ram-client-0.75	0.18	1	0.31
ram-client-0.375	0.19	1	0.31
ram-server-3	0.26	0.33	0.29
ram-server-1.5	0.27	0.33	0.29
ram-server-1.125	0.33	0.49	0.39
ram-server-0.75	0.19	1	0.32
cpu-client-150	0.21	0.24	0.22
cpu-client-75	0.38	0.4	0.39
cpu-client-56.25	0.34	0.88	0.49
cpu-client-37.5	0.27	0.97	0.43

Table A.1.: Performance of all configurations for strict classification of flaky and stable

List of Figures

5.1. Distribution of CPU usage for server and client during regular test execution 21
5.2. Distribution of RAM usage for server and client during regular test execution 22
5.3. Fail rates per test per configuration - brighter colors indicate higher fail rates 23
5.4. Average fail rates per resource 24
5.5. Effort of detection for regular and resource-limited rerun $S(n, 0.95, T)$. 26

List of Tables

5.1. Maximum usage parameters used in the limitation configurations . . . 22
5.2. Performance of the top five configurations and baselines 25
5.3. Comparing detection effort for *Rerun* and our approach in terms of number of runs required for detection 28
5.4. Performance of the top five configurations and baselines for a more inclusive classification of flaky tests 30

A.1. Performance of all configurations for strict classification of flaky and stable 39

Acronyms

SUT System Under Test

CI Continuous Integration

CD Continuous Delivery

OD Order-Dependant

NOD Non-Order-Dependant

cgroups Control Groups

Bibliography

[1] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 830–840, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338945.

[2] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An Empirical Study of Flaky Tests in Python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Porto de Galinhas, Brazil: IEEE, Apr. 2021, pp. 148–158, ISBN: 978-1-72816-836-4. DOI: 10.1109/ICST49551.2021.00026.

[3] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "What do developer-repaired Flaky tests tell us about the effectiveness of automated Flaky test detection?" In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22, New York, NY, USA: Association for Computing Machinery, Jul. 19, 2022, pp. 160–164, ISBN: 978-1-4503-9286-0. DOI: 10.1145/3524481.3527227.

[4] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, New York, NY, USA: Association for Computing Machinery, Nov. 11, 2014, pp. 643–653, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635920.

[5] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An Empirical Analysis of UI-Based Flaky Tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1585–1597. DOI: 10.1109/ICSE43902.2021.00141.

[6] D. Silva, L. Teixeira, and M. d'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2020, pp. 301–311. DOI: 10.1109/ICSME46990.2020.00037.

[7] M. Barboni, A. Bertolino, and G. De Angelis, "What We Talk About When We Talk About Software Test Flakiness," in *Quality of Information and Communications Technology*, A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, and R. Pérez-Castillo, Eds., ser. Communications in Computer and Information Science, Cham: Springer International Publishing, 2021, pp. 29–39, ISBN: 978-3-030-85347-1. DOI: `10.1007/978-3-030-85347-1_3`.

[8] M. Harman and P. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2018, pp. 1–23. DOI: `10.1109/SCAM.2018.00009`.

[9] "Google Testing Blog: Where do our flaky tests come from?" (), [Online]. Available: `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html` (visited on 03/13/2023).

[10] J. Micco, *The State of Continuous Integration Testing @Google*, 2017.

[11] F. J. Lacoste, "Killing the Gatekeeper: Introducing a Continuous Integration System," in *2009 Agile Conference*, Aug. 2009, pp. 387–392. DOI: `10.1109/AGILE.2009.35`.

[12] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing China: ACM, Jul. 10, 2019, pp. 101–111, ISBN: 978-1-4503-6224-5. DOI: `10.1145/3293882.3330570`.

[13] "MethodSorters (JUnit API)." (), [Online]. Available: `https://junit.org/junit4/javadoc/4.12/org/junit/runners/MethodSorters.html` (visited on 04/07/2023).

[14] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "UI-Based Flaky Tests Dataset," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Madrid, ES: IEEE, May 2021, pp. 234–235, ISBN: 978-1-66541-219-3. DOI: `10.1109/ICSE-Companion52605.2021.00108`.

[15] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Xi'an, China: IEEE, Apr. 2019, pp. 312–322, ISBN: 978-1-72811-736-2. DOI: `10.1109/ICST.2019.00038`.

[16] A. Gambi, J. Bell, and A. Zeller, "Practical Test Dependency Detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2018, pp. 1–11. DOI: `10.1109/ICST.2018.00011`.

[17] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: Association for Computing Machinery, Jul. 21, 2014, pp. 385–396, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610404.

[18] "How do you test your tests?" Engineering at Meta. (Dec. 10, 2020), [Online]. Available: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/ (visited on 04/10/2023).

[19] "Runtime options with Memory, CPUs, and GPUs," Docker Documentation. (Apr. 6, 2023), [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/ (visited on 04/07/2023).

[20] "Control Groups — The Linux Kernel documentation." (), [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html (visited on 04/10/2023).

[21] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Jan. 31, 2022, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3476105.

[22] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 433–444. DOI: 10.1145/3180155.3180164.

[23] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related Flaky Test Detection in Android apps," version 3, 2020. DOI: 10.48550/ARXIV.2005.10762.

[24] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and Ranking Flaky Tests at Apple," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Oct. 2020, pp. 110–119.

[25] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, Aug. 12, 2019, pp. 545–555, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338925.

[26] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2022, pp. 244–255. DOI: 10.1109/ICST53961.2022.00034.

[27] "Test Retries | Cypress Documentation." (), [Online]. Available: https://docs.cypress.io/guides/guides/test-retries (visited on 03/26/2023).

[28] "Swap Management." (), [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand014.html (visited on 04/11/2023).

[29] "Out Of Memory Management." (), [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand016.html (visited on 04/11/2023).

[30] "CFS Bandwidth Control — The Linux Kernel documentation." (), [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-bwc.html (visited on 03/26/2023).