Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--25/001--SE

Analyzing Root Causes and Smells of Test Flakiness by Simulating Resource Usage

– A study about how system resource limitations can induce flaky behavior

---

Att Analysera Rotorsaker och Lukter av Testinstabilitet genom att Simulera Resursanvändning
– En studie om hur resursbegränsningar i system kan orsaka testinstabilitet

Martin Jonson (marjo886)
Simon Törnqvist (simto860)

Supervisor : Xin Sun
Examiner : Daniel Varro

External supervisor : Dimitris Rentas

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

Flaky tests, which intermittently pass or fail when no changes have been made to the code, are a significant challenge in regression testing. Such flakiness affects the reliability of test outcomes, forcing developers to debug false alerts. Current research suggests that some test flakiness is associated with computational resource availability. In this thesis, we investigate the impact of computational resource availability, specifically CPU time, on the frequency of test flakiness in an industrial C++ codebase. We conduct experiments by rerunning tests under simulated CPU usage to quantify this impact. Furthermore, we analyze test source code characteristics and known flakiness root causes to provide insight.

The findings reveal that CPU usage significantly influences certain types of flakiness. While concurrency-related flakiness is varyingly affected, flakiness associated with waiting asynchronously is shown to be completely dependent on CPU usage. Additionally, the study suggests that maintainability and thread sleep-related software characteristics could serve as predictors of resource-affected flakiness. These insights highlight the importance of considering computational resources in test design and suggest potential areas for improving test reliability through resource management and code analysis practices.

Acknowledgments

This endeavor would not have been possible without Dimitris Rentas, our supervisor at Ericsson, who helped us continuously improve our work throughout the project. We also want to express our gratitude to our examiner at the university, Daniel Varro, for his invaluable involvement throughout the project, and to our supervisor, Xin Sun, for his persistent support.

We are thankful for the great people at Ericsson that have been involved in this thesis work by accommodating us, suggesting technical solutions and providing expert knowledge. Their warmth and dedication are matched only by their exceptional competence.

We would like to acknowledge our friends and families, who have supported us during this project as well as the six years of studies leading up to it. Lastly, we want to mention the UPP group that was our extra home for five years.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, the motivation behind this work is stated in the context of software development, the industry and the research community. The aim is proposed as the intended contribution of this work to both the industry and to the research in this area. The research questions and the delimitations of this work are presented to specify what this thesis includes.

1.1 Motivation

Software testing is widely used in the industry to ensure high-quality software. Developers need reproducible and reliable results from testing to trust it completely, especially to avoid regressions during development. A threat to this trust is test flakiness, or flaky tests, which are non-deterministic tests that may pass or fail independently of changes to the executed code [1]. 91% of developers in the industry deal with flaky tests at least a few times a year and 24% say that they deal with them every week [2]. When a flaky failure occurs, it can lead to unnecessary debugging time for developers trying to solve an issue that is not related to the changed code. The process of frequently integrating changes to a project, called continuous integration, can be severely impacted by flaky tests. This is because regression testing pipelines are commonly used in continuous integration to ensure that previous features do not break when new changes are added, and these pipelines need to pass to be allowed to merge the changes. This can take several hours in large code bases, which makes it very costly when flaky failures occur. Flaky tests can also cause developers to ignore the results of test suites if the trust in the testing infrastructure is strained [3].

Many tools have been developed in the last 20 years to detect flaky tests but there is no catch-all method [4]. Recently there has been work investigating how resource availability at runtime can change the outcomes of tests. The term resource-affected flaky test (RAFT) has been introduced as a subcategory of flaky tests [5], and tools have been created that add noise to testing environments to force failures to occur more frequently [6, 7]. This thesis builds upon these recent findings to establish connections between resource usage and test flakiness. It also introduces resource-independent flaky test (RIFT) to the nomenclature.

Throughout the years, a wide variety of root causes of test flakiness have been discovered [1, 8, 9, 10]. Their abundance and diversity make them complex to discover and solve. When software systems become larger and more complex, these root causes are likely to appear. Some of the most common ones are: *Async Wait* in asynchronous programming, *Con-*

currency in multithreaded applications and *Test Order Dependency* which can appear in large test suites. Understanding these root causes and their relationship with resource usage could enable improved flakiness detection tools in the future.

Current research suggests that test flakiness can be predicted, to some extent, based on source code characteristics without even running the tests [4, 11]. Code metrics that measure program complexity, such as *Lines of Code* and *Cyclomatic Complexity*, differ between flaky and non-flaky tests in some projects [12], possibly enabling prediction. Additionally, test smells — defined as test code anti-patterns [13] — have been shown to predict flakiness [11]. We combine code metrics and test smells with the concept of resource-affected flakiness to further investigate the feasibility of flakiness prediction.

1.2 Aim

Previous research has proven that limited resource availability can lead to intermittent test failures [6]. Some flaky tests are affected by the resource availability while other flaky tests' failure rates remain consistent over different stress configurations [5]. The aim of this thesis is to apply simulated resource usage to closed-source industry tests, to expand upon the previous research. Furthermore, it aims to find connections between the available resources at the time of the test run and two aspects: (1) what causes tests to fail and (2) what implementation anti-patterns for test code exist in the codebase. In this way, it lays the groundwork for further exploration in both industry and the research community.

1.3 Research questions

RQ1: How does computational resource availability correlate with test flakiness?

RQ2: How do specific root causes of flaky tests and computational resource availability contribute to flakiness?

RQ3: How do test smells, code metrics and computational resource availability predict test flakiness?

1.4 Delimitations

The resource limitation for this work is focused on CPU usage, which has been shown to affect flaky tests the most [5]. Other resource considerations such as memory, disk space and network connection are out of scope for this study.

The tests used in the experiments are limited to a part of an industry C++ codebase. This complements existing research which has been conducted on open-source projects [6, 7].

While these delimitations provide a practical and focused approach, it may limit the generalizability of the findings. Future work could extend the experiments to include other resources and diverse codebases to further validate and expand the results.

1.5 Thesis outline

This thesis is divided into the following chapters to enhance the reading experience. The background is introduced in Chapter 2 and describes the underlying theory behind this thesis. Chapter 3 puts this thesis in the context of the past and current work in this research area. The methods that are used to answer the research questions in this thesis are described in Chapter 4. The results of the conducted experiments are revealed in Chapter 5 and discussed along with the method in Chapter 6. The key findings of the thesis are presented along with a selection of ideas for future work in Chapter 7.

2 Background

This chapter introduces the reader to areas relevant for understanding the contents of this thesis. We begin with introducing software testing definitions, practices and anti-patterns (Section 2.1). Secondly, we describe flaky tests and their root causes (Section 2.2). We then briefly introduce continuous integration (Section 2.3). Finally, we present important computer architecture concepts related to computer resources and non-determinism (Section 2.4).

2.1 Software testing

Software testing is the practice of evaluating software to find errors and ensuring that the software behaves as it should [14]. Test code is commonly defined by its *test cases*, where each test case tests a different attribute of the software [14, 15, 16, 17]. Test cases may be grouped into *test suites* when they share scope or test similar aspects [15, 16, 17]. When a test requires set up or tear down logic, possibly shared with other tests, a *test fixture* (e.g., class, module, component) can be created to encapsulate and manage that logic [15, 16, 17]. Listing 2.1 illustrates an example test case. The test case tests that the function *Foo* writes the text "Hello World" to the passed file. The fixture module exposes a file that can be safely written to, reusable by other test cases.

```
1  -- Fixture: Set up safe out file
2  Out_File : File;
3
4  procedure Set_Up is
5      Out_File := Open("tmp");
6  end Set_Up;
7
8  -- Test case: Expect Foo to write "Hello World" to file
9  with Fixture;
10 procedure Test_Foo is
11     Fixture.Set_Up;
12     Foo(Fixture.Out_File);
13     Expect(Read(Fixture.Out_File) = "Hello World");
14 end Test_Add;
```

Listing 2.1: Example test case and fixture

There are three common levels of scopes of software testing: *unit testing*, *integration testing* and *system testing* [18, 19].

- **Unit testing** is the lowest level of testing, and aims to test the "units" within a module [19]. The definition of a unit differs in literature. It can be referred to as a statement [19], block of code [20], function [20, 21], individual component [18] or collection of components [18]. Commonly, unit testing tests the selected unit in isolation of other parts of the system [18, 19, 20, 21].
- **Integration testing** tests the behavior of selected modules when they are combined into a working program [19], and specifically targets the interaction between the modules over their inner details [18, 19].
- **System testing** tests the entire system against its higher-level, non-implementation specific requirements, such as whether the system behaves as described in its specification and how it performs against performance requirements [22].

In addition to these levels of scope, software testing is further divided into two primary categories: *functional testing* and *structural testing* [18].

- **Functional testing** (aka. black-box testing) tests that the software behaves as intended, independently of its source code structure. The attributes tested are based on the software specification or the software's behavior [18].
- **Structural testing** (aka. white-box testing) instead tests based on internal details of the source code [23, 18], such as data flow or which statements are executed [23].

2.1.1 Regression testing

Regression testing is the practice of retesting new software versions to ensure new changes do not break existing functionality [18]. New features and improvements are called progressions, while the breaking of existing functionality is called a regression [24].

Regression testing gives developers confidence that their changes are safe to commit to the mainline version, or helps them immediately find problems in their code [25]. Furthermore, regression testing gives an indication of which commit that broke the tests [25]. This helps developers focus on debugging the changes that resulted in the regression test failure and ignore unrelated code, instead of first having to try to find the problematic code [25]. Regression testing may also be used as a gateway in a continuous integration build system to reject faulty commits and maintain the integrity of the mainline version [26]. Without automated regression testing, developers themselves have to figure out when the bug was introduced and what code changes might have caused it [25]. Furthermore, the absence of regression testing makes teams afraid of making necessary changes and slows down development [25].

In order for regression testing to be effective, a test should only fail if the error was introduced by the changes under investigation. However, the presence of tests with non-deterministic outcome, also known as flaky tests, can severely reduce the reliability of regression testing [25, 1, 12, 11, 27].

2.1.2 Test smells

Test smells are a set of known test code patterns with negative impacts, such as code obscurity or non-deterministic test behavior [13]. There are many test smells in literature (e.g., 66 in [28]). In Table 2.1 we present the original 11 ones by Van Deursen et al. [13], and some more test smells relevant to the thesis[1].

[1]Some descriptions in the table are generalized from JUnit-specific details in [13] and [29].

Table 2.1: Test smells, adapted from Van Deursen et al. [13] and Meszaros [30]

Mystery Guest	Test code that uses external resources, such as directories, files or databases, without explicitly setting them up or allocating them. Mysterious external resources make it hard to understand what is being tested from the test code itself, making it harder to pinpoint the root cause of a test failure. [13]
Resource Optimism	Test code that makes optimistic assumptions about external resources, such as the state of directories, files or databases. Assuming the state of an external resource can cause the test to fail due to said resource not behaving as expected, instead of only failing when the code under test is at fault. [13]
Test Run War	Test code that interferes with other tests when run simultaneously. Typically refers to interfering resources, for example two tests creating temporary files with colliding names. [13]
Sleepy Test	Test code that explicitly causes thread sleeping [29]. Sleeping for a fixed amount of time in order to wait for a resource may cause flaky behavior, and should be replaced by synchronous waiting directives [1].
Conditional Test Logic	Test code that contains control statements such as if and for [29, 30]. Such tests are hard to understand, since different executions follow different control paths [30]. Furthermore, if the conditions are not deterministic with regards to the input data, then the test becomes non-deterministic [30].
General Fixture	Test code with too general fixtures, where each test case only needs part of a fixture. General fixtures are harder to read and understand than smaller, specific fixtures. Fixtures with general setup logic also risks slowing down the tests, making developers stop running the tests. [13]
Eager Test	A test case that tests too many different behaviors of the test subject. It should be easy to understand the purpose of a test by reading it, but eager tests are too general. An eager test should be separated into multiple test cases with meaningful names. [13]
Lazy Test	A test case that tests the same functionality as other test cases, but with different criteria. For example, multiple test cases testing the same instance method using the same fixture, but check the values of different instance variables. Such test cases should be joined into a single test case. [13]
Assertion Roulette	Test code with unexplained assertions, making it hard to understand which assertion failed when reading the test output. [13]
Indirect Testing	Test code that tests out of scope functionality. For example, a test case intended to specifically test one class only, but instead tests methods of other classes as well. Indirect tests are harder to understand and debug. [13]
For Testers Only	Production code that includes functionality only used for testing. Testing functionality should only be present in test code. [13]
Sensitive Equality	Test code that implements its own equality operations for production classes, that might break in the future. For example, if a test case uses the comparison between a string literal and the string serialization of an object as equality, then changing the string serialization method causes the test to fail. If test code depend on the equality check of a production class, an equality method should be introduced in the production code. [13]
Test Code Duplication	As the name implies, duplicated code in multiple test cases. For example, duplicated fixture setup or teardown logic that should be extracted to the common fixture logic. [13]

Some test smells can indirectly lead to unexpected test behavior. Mystery Guest, Resource Optimism and Test Run War all refer to improper ways of handling external resources, and can indirectly cause non-deterministic test behavior if the external resources themselves are non-deterministic with regards to the test input [31]. Sleepy Test may inadvertently cause flakiness when the reason for sleeping is to wait for a resource, as in the Async Wait flakiness root cause (Section 2.2.1). Furthermore, thread sleeping may imply multithreading, which in itself can introduce non-deterministic instruction interleaving and data races (Section 2.4.6) that in turn cause Concurrency related flakiness (Section 2.2.1).

Other test smells cause obscurity and maintainability issues, indirectly leading to developer mistakes. General Fixture and Eager Test refer to obscured code by bloating, the former in fixtures and the latter in test code, which may make it harder to understand the tests and identify mistakes [30]. Lazy Test, Assertion Roulette, Indirect Testing and Test Code Duplication similarly obscure code not by bloating, but by poorly described or misplaced code.

Conditional Test Logic may cause a mix of code obscurity and non-determinism issues. Complicated control flow makes it hard to follow the purpose of the test and verify its correctness, requiring the logic of the test itself to be tested [30]. Furthermore, a test should test the same behavior for every execution, but Conditional Test Logic may instead cause the test to follow a different path between different executions [30], in turn causing different results for the same code under test.

2.2 Flaky tests

Flaky tests are tests that non-deterministically pass and fail without changes to the test code or code under test [1, 4]. The presence of flaky tests is often referred to as test flakiness [1, 4]. When a regression test fails it should imply that the commit under test caused the failure. However, a flaky test can by definition start to fail at any time, causing false alarms [4].

2.2.1 Flakiness root causes

Luo et al. [1] introduce a classification of test flakiness root causes in their empirical study, with 10 different root causes. Since then, a number of studies have used and changed these to fit their specific needs. Parry et al. [4] summarize recent studies and finds 14 categories in 2021. Table 2.2 shows the original 10 root causes identified by Luo et al. [1], and their relative frequencies in different studies. Root causes not appearing in a particular study are marked with a dash. Each root cause is then explained in the following subsections.

Table 2.2: Flakiness root causes relative frequencies in studies

Root Cause	Luo et al. [1]	Eck et al. [8]	Lam et al. [9]	Hashemi et al. [10]
Async Wait	0.45	0.22	0.78	0.21
Concurrency	0.20	0.26	0.14	0.20
Test Order Dependency	0.12	0.11	0.00	—
Resource Leak	0.07	0.07	0.05	0.03
Network	0.06	—	0.14	0.13
Time	0.03	0.02	0.04	0.03
IO	0.02	—	0.05	—
Randomness	0.02	0.02	0.05	—
Floating Point Operation	0.02	0.03	0.02	—
Unordered Collections	0.01	—	0.00	—

Async Wait

Test flakiness caused by Async Wait occurs when a test does not properly wait for asynchronous operations to complete before continuing [1]. Depending on how much time the setup takes, the test may pass or fail [1]. Listing 2.2 shows an example of an Async Wait flaky test, where an asynchronous setup step *Start_Server* is required. A fixed time sleep is used to wait for the setup, but if it for any reason takes more than one second then the test fails.

```
1  procedure Test_Server_Request is
2      Start_Server;
3      Sleep(1);
4      Response := Make_Request;
5      Assert_Ok(Response);
6  end Test_Server_Request;
```

Listing 2.2: Example of Async Wait flaky test

Concurrency

A test that intermittently fails due to non-deterministic multithreading is classified with Concurrency as its flakiness root cause [1]. Unlike Async Wait, the Concurrency category may refer to non-determinism in the code under test in addition to the test code [1]. Since multithreaded code may inherently behave non-deterministically, the non-determinism in the code itself does not have to be a bug [1]. However, a test that fails due to incorrectly assuming said code is deterministic becomes flaky [1]. Listing 2.3 shows an example of a Concurrency flaky test, caused by an erroneous data race in the code under test. If two threads both execute line 7 before either execute line 8, they will both read the same value into *Previous_Tasks_Completed*. This causes both threads to "increment" *Tasks_Completed* into the same value, effectively incrementing by one instead of two. In this example, threads should be prevented from simultaneously entering the "critical section" from line 6, using synchronization directives (Section 2.4.6).

```
1  Tasks_Completed := 0;
2
3  procedure Worker is
4      Perform_Task;
5
6      -- Critical section
7      Previous_Tasks_Completed := Tasks_Completed;
8      Tasks_Completed := Previous_Tasks_Completed + 1;
9  end Worker;
10
11 procedure Test_Worker is
12     for I in 1..10 do
13         Spawn_Thread(Worker);
14     end for;
15     Join_Threads;
16     Assert(Tasks_Completed = 10);
17 end Test_Worker;
```

Listing 2.3: Example of Concurrency flaky test

Test Order Dependency

A test that intermittently fails depending on the order in which it is run with the rest of the test suite is classified with Test Order Dependency as flakiness root cause [1]. Test order dependencies occur when tests within a test suite depend on shared state that is not properly managed [1]. Shi et al. [32] propose two types of order-dependent tests: *victim* and *brittle*. An

order-dependent victim test fails because a previous test polluted the shared state, without either test properly resetting the state. This type of test passes when run in isolation, but fails when run after the polluting test. An order-dependent brittle test fails when run in isolation, but passes when run after a test that helps set up the state [32]. Listing 2.4 shows an example of a Test Order Dependency flaky test. *Test_Logout* requires there to be a logged in session, which happens only if *Test_Login* runs before. If the test order is changed, or *Test_Login* is removed, then *Test_Logout* fails. In this example *Test_Logout* is a brittle test.

```
procedure Test_Login is
    Response := Login;
    Assert_Ok(Response);
end Test_Login;

procedure Test_Logout is
    Response := Logout;
    Assert_Ok(Response);
end Test_Logout;
```

Listing 2.4: Example of Test Order Dependency flaky test

Resource Leak

A test that is flaky due to the code under test not properly releasing acquired resources is classified with Resource Leak as flakiness root cause [1]. The resource in question may be anything external that the code under test is responsible for managing, e.g., memory or database connections [1, 8]. A resource leak does not necessarily manifest in a single test [25]. Instead, any test using the resource in question may fail once the limit is reached [25]. Listing 2.5 shows a test of function *Memory_Expensive_Operation*, that allocates memory without releasing it. If, for instance, other tests also make use of memory-leaking methods, then memory could run out. This would in turn trigger non-deterministic failures. Furthermore, running out of memory could cause other tests in the suite to fail.

```
function Memory_Expensive_Operation(A : Integer) return Integer is
    Mem := Allocate_Much_Memory;
    return Some_Operation(A, Mem);
    -- Forget to deallocate
end Memory_Expensive_Operation;

procedure Test_Memory_Expensive_Operation is
    Assert(Memory_Expensive_Operation(10) = 100);
    Assert(Memory_Expensive_Operation(42) = 1);
    Assert(Memory_Expensive_Operation(1) = 42);
end Test_Memory_Expensive_Operation;
```

Listing 2.5: Example of Resource Leak flaky test

Network

A test that intermittently fails due to a remote or local connection failure is classified as a Network flaky test [1]. The failure may be caused due to (inter)net problems, a remote dependency being unavailable or bad socket management. The fault may be in the code under test or in the test code alike. Listing 2.6 shows an example test of function *Fetch*, which connects to the internet to download results. This could fail at any time due to networking problems on the test machine.

```
1 procedure Test_Fetch is
2     URL := "https://example.com";
3     Result := Fetch(URL);
4     Assert(Result.Status_Code = 200);
5 end Test_Fetch;
```

Listing 2.6: Example of Nework flaky test

Time

A test that intermittently fails because it depends on the local system clock is classified with Time as flakiness root cause [1]. Since all calls to the system clock are non-deterministic, any test or code under test using it risks becoming non-deterministic as well [25]. For example, if a test assumes that two retrieved dates are on the same day but fails because the clock just hit midnight then the test is Time flaky [8]. Fowler [25] argues that Time flakiness is so common that one should prohibit all direct calls to the system clock. We actually had a Time-related bug during implementation, where a script began to crash after the change from summer to winter time, causing prolonged and frustrated debugging. Listing 2.7 shows an example test that could fail at midnight, since *Current_Hour* going from 23 to 0 while *Perform_Operation* is ongoing would invalidate the assertion on line 5.

```
1 procedure Test_Performance is
2     Start_Time := Current_Hour;
3     Perform_Operation;
4     End_Time := Current_Hour;
5     Assert(0 <= End_Time - Start_Time <= 1);
6 end Test_Performance;
```

Listing 2.7: Example of Time flaky test

IO

A test that intermittently fails due to non-deterministic I/O operations is classified with IO as flakiness root cause [1]. For example, if the disk is or becomes full, the test that happens to be running might fail [4]. Listing 2.8 shows an example test that validates the creation of a file, which could non-deterministically fail if there is no disk space.

```
1 procedure Test_Fetch is
2     File := Open("My_File.txt");
3     File.Write("Hello");
4     File.Close;
5     Assert(File_Exists("My_File.txt"));
6 end Test_Fetch;
```

Listing 2.8: Example of IO flaky test

Randomness

A test that intermittently fails due to misusing randomly generated values is classified with Randomness as its flakiness root cause [1]. For example, a test might fail specifically when the number one is generated but pass in all other cases. Listing 2.9 shows an example test that validates a division operation with random parameters. Even if the division is perfect, randomness could cause a non-deterministic failure if *Denominator* is assigned zero.

```
1 procedure Test_Division is
2     Numerator := Random;
3     Denominator := Random;
4     Result := Division(Numerator, Denominator);
5     Assert(Numerator = Result * Denominator);
6 end Test_Division;
```

Listing 2.9: Example of Randomness flaky test

Floating Point Operations

Floating point operations can cause non-determinism, for example due to rounding, underflow and overflow behavior. A test that intermittently fails due to such behavior is classified with the Floating Point Operations as its flakiness root cause [1]. Listing 2.10 shows an example test that could randomly fail due to rounding errors. On line 3, the order of operations could cause cancellation, making A being assigned near-zero. But on line 4, the order prevents cancellation, assigning 1 to B. This makes the assertion fail. However, depending on floating point precision and compiler optimization flags, the cancellation is uncertain, meaning that failure could be non-deterministic for the same code.

```
1 procedure Test_Add is
2     A := (1 + 1e20) - 1e20;
3     B := 1 + (1e20 - 1e20);
4     Assert(A = B);
5 end Test_Add;
```

Listing 2.10: Example of Floating Point Operations flaky test

Unordered Collections

Assuming the order of an unordered collection (e.g., set) when iterating it may lead to non-deterministic behavior. A test that intermittently fails due to this is classified with Unordered Collections as root cause of the flakiness [1]. Listing 2.11 shows an example test expecting an unordered set's iterative order to be the same as that of its passed constructor values. If this is not guaranteed by the language implementation, then the test could become flaky.

```
1  procedure Test_Set is
2      S := Unordered_Set(1,2,3);
3      Result := "";
4
5      for Value in S loop
6          Result += Value;
7      end loop;
8
9      Assert(Result = "123");
10 end Test_Set;
```

Listing 2.11: Example of Unordered Collections flaky test

2.3 Continuous integration

Continuous integration is the practice of developers frequently integrating their changes while maintaining a stable codebase. It is performed by having a shared code repository with a mainline version that developers can commit changes to and pull other developers' changes from. For every committed change, a build is carried out to ensure that there were no integration errors or mistakes. A continuous integration server can be used to automatically

build and test committed changes, providing fast feedback to developers. One goal of continuous integration is to reduce the time spent for integrating the contributions of individual developers and increase the shipping speed. [33, 34]

An alternative to continuous integration is to have developers work on their features and not integrate until one or multiple features are done. This may lead to longer and more unpredictable integration cycles. [34]

2.4 Computer architecture

This section introduces key concepts in computer architecture necessary for understanding the content of this thesis. For accessibility, some parts are intentionally simplified. While not every computer works exactly as described here, the descriptions are applicable to most modern systems, including those used for the experiments of this thesis. Key concepts described here are *memory* (2.4.1), *processor* (2.4.2), *operating system* (2.4.3), *multiprogramming* (2.4.4), *multiprocessing* (2.4.5) and *multithreading* (2.4.6).

2.4.1 Memory

The memory of a computer is divided into a hierarchy of memory levels with different properties. Figure 2.1 shows a high level abstraction of the memory hierarchy. Faster memories are reserved for the most frequently used and time critical data, while larger memory types are used for infrequent data. The reasoning behind this is that fast memory is expensive. A proper memory hierarchy can reach satisfactory performance while diluting fast memory with cheaper, slower memory. [35]

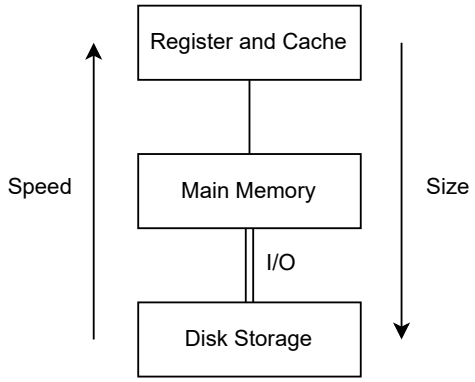


Figure 2.1: Computer memory hierarchy

The *register* and *cache* memories are the fastest but smallest memories. They are tightly coupled with a single processor, although some caches might be shared across multiple processors on a single computer. These types of memories are not considered in the method of this thesis, but could be relevant as they impact non-deterministic latency. The *main memory*, also referred to as just the memory, is where program data is stored, and is shared by all processors. While slower than the registers and caches, the main memory is dramatically faster than disk storage. The main memory is divided into blocks, each block with its own physical address used to access it. Typically, each variable in a program has its own such address. *Disk storage* is the largest of the memory types. To access a part of disk storage, an input/output (I/O) operation must be performed to transfer data between main memory and disk storage. Hennessy and Patterson [35] make the comparison of 5–10 milliseconds required for accessing disk storage with 50–100 nanoseconds required for accessing the main memory, which is a difference of factor 100,000. Although the comparison was given in 2011 and hardware capa-

bilities rapidly evolves over time, this relation of memory access times is still highly relevant in todays computers. [35]

2.4.2 Processor (CPU)

The main component of the computer is its *processor*, also known as its Central Processing Unit (CPU). The processor is responsible for interpreting and executing instructions located in memory. A *single-core processor* can execute instructions from a single source at a time. However, processors often have multiple cores each, allowing them to execute instructions from multiple sources in parallel (Section 2.4.5). [35]

When a processor has no instructions to execute, because the running process is waiting or there are currently no tasks to process, it becomes idle. An important resource metric for a processor is *CPU time*, which is the time a processor spends executing instructions, excluding idle time and time waiting for I/O operations. *CPU utilization* is the CPU time relative to idle time and wait time. A higher CPU utilization means that more instructions are executed per time unit, meaning that more of the CPU's potential is realized. [35]

2.4.3 Operating system

A computer may have many user programs running simultaneously, managed by the *operating system* [35]. A *process* is an instance of a running program and its state, including its allotted memory [35]. Different processes are prevented to access each others memory through memory virtualization [35]. The primary method of memory virtualization is paging [35]. A page-based virtual memory is divided into several pages, and each process is given a set of exclusive pages that no other process may access [36]. The processes can only refer to memory through virtual addresses, that are translated by the operating system into physical memory addresses [36]. This allows program code to treat memory as contiguous even though it is actually fragmented [36]. It also allows the operating system to allocate more pages than what the main memory can fit, and store the extra pages in disk storage [36]. When a process wants to access a page in disk storage, the operating system must first *swap* the page into main memory and swap another page out [36]. The process cannot continue to execute until the swap is done [36].

2.4.4 Multiprogramming

Multiprogramming is the concept of running several processes concurrently on the same computer, by frequently switching which process is currently running in a so called *context switch* [35]. By allowing all processes to progress a little bit at a time, they are perceived to be executed in parallel [35]. Note, however, that this is not "true parallelism" (Section 2.4.5). Furthermore, context switching allows the computer to reduce idle time in a CPU by switching a waiting process with one that has instructions to execute, effectively "hiding" latency caused by slow I/O operations and process waiting [35]. A process can actively decide to enter a *blocked* state, telling the operating system to not run it until a certain event [36]. This is called *sleeping* or blocking, and can be used to coordinate processes. For multithreaded processes (Section 2.4.6), sleeping is performed on thread-level [36]. Figure 2.2 shows two example executions of two processes on a single core. In both examples, *process 1* waits after running for a short time, maybe due to sleeping or waiting for I/O, and then continues until done. In the upper example, *process 2* is not allowed to execute until process 1 is completely done, causing the wait to affect the total time. But in the lower example, there is a context switch immediately when process 1 begins to wait, allowing process 2 to complete and reducing the total time needed for both processes to complete.

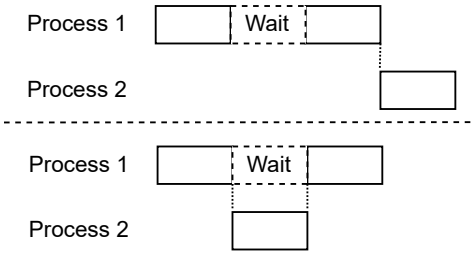


Figure 2.2: Example of multiprogramming concurrency

It should be noted that context switches themselves introduce overhead, and can negatively affect performance when used too frequently [36]. For instance, if the memory of the newly switched-in process has been page swapped into disk storage, a disk read must be performed to swap the page back in, potentially introducing delay [36]. Furthermore, context switching introduces non-deterministic instruction interleaving [36]. Where a "singleprogramming" execution reads from a deterministic sequence of instructions, a multiprogramming execution interleaves instructions from multiple processes into a non-deterministic sequence [36]. Consider the program in Listing 2.12. Executing the program two times in sequence yields the output "1 2 3 1 2 3". However, executing two instances of the program concurrently while printing to the same destination can yield many more outputs. For example "1 2 3 1 2 3" and "1 1 2 2 3 3" are both valid outputs, depending on how the operating system decides to context switch. Moreover, context switching not only causes non-deterministic instruction interleaving, but also non-deterministic latency in-between instructions since the operating system can pause the execution of a process at any time.

```
1 procedure Print_123 is
2     Print("1");
3     Print("2");
4     Print("3");
5 end Print_123;
```

Listing 2.12: Example Sequential Program

2.4.5 Multiprocessing

Multiprocessing is the concept of executing multiple processes in parallel. It refers to true parallelism, and requires utilization of multiple CPU cores. Every core has a maximum CPU time it can spend on executing instructions in any given time frame. A higher number of cores increases the maximum CPU time of the entire computer, increasing potential instruction throughput. It should be noted however, that adding more cores does not guarantee a linear increase in performance [35].

2.4.6 Multithreading

Processes can not only run concurrently and in parallel with each other, but each process may in turn execute in several *threads*. Similar to processes in multiprogramming, each thread executes its own set of instructions concurrently, and is switched by the operating system with so called *thread switching*. However, since threads execute in the same process they all have access to the same memory. Sharing memory enables thread switching to cause less overhead than context switching. Figure 2.2 actually describes multithreading and not only multiprogramming, just replace the processes with threads. [35]

Sharing data between threads is a potential solution for coordinating them, but risks introducing *data races*. A data race occurs when multiple threads access the same data in an

unsafe manner, which may cause unpredictable results depending on the non-deterministic order of thread execution. Non-determinism related to multithreading is caused by the same reasons as for multiprogramming: instruction interleaving and latency. [35]

To solve data race bugs, one can apply *synchronization* methods such as *locks* to ensure that only one thread can access a variable at a time [37]. However, this can in turn introduce *deadlock* bugs where threads infinitely wait for each other to release their locked variables [37]. Writing concurrent code is hard, and concurrency related bugs are easily introduced but difficult to debug [37]. For example, the "Dirty COW" vulnerability was a race condition bug present in the well renown Linux Kernel for over a decade, that could be exploited by a user to elevate their privileges [38]. Lu et al. [37] find many concurrency related bugs to be caused only by the ordering of two threads, meaning that those bugs can be manifested by enforcing a particular execution order of those two threads. Furthermore, they argue that execution under a high workload of many threads increases context switch frequency, which in turn increases the probability of the buggy thread order being executed.

Summary: This chapter introduced key knowledge areas essential for understanding the content of this thesis. An overview of software testing was provided along with specific patterns and anti-patterns in regression testing. Test flakiness and its root causes was discussed, supported by clear code examples. Additionally, the chapter covered continuous integration and an introduction to computer architecture to highlight the significance of computational resources and non-determinism. The next chapter, Related Work, will provide an overview of the work that has been carried out in the same area of research as this thesis.

---

3 Related Work

This chapter explores related research in this field. Section 3.1 presents techniques for detecting test smells, while Section 3.2 introduces some methods for identifying flaky tests.

3.1 Test smells

There has been some work in the area of detecting test smells [28]. This section covers several tools that have been developed to find the test smells specified in Section 2.1.2. Each tool and the individual test smells that they aim to find are listed in Table 3.1. The smells that this work aims to analyze are marked as bold.

TestQ is a tool developed by Breugelmans and Van Rompaey [39] that uses metrics to detect test smells. It builds an Rigi Standard Format (RSF) model based on the source code and queries for test smells [39].

Bavota et al. [31] created a high recall tool to find test smells in JUnit tests. The tool analyzed Java code with a rule based detection system and was tested on both open-source and industry data [31]. The rules were chosen to be simple and to overestimate the number of test smells found. An example is the rule for Eager Test: "JUnit classes having at least one method that uses more than one method of the tested class" [31]. The unnamed tool prioritized recall over precision and therefore needed manual examination to verify results [31].

TestHound is a static analysis test smell detection tool created by Greiler et al. [40]. Their focus was on test fixtures in Java code, and they evaluated the tool on both industry and open-source projects [40]. **TestEvoHound** was an improvement on this tool that analyzed tests for different versions in the version control system [41].

TASTE is a tool developed by Palomba et al. [42] to find test smells using textual information, improving on the work of metric based approaches such as Greiler et al. [40]. TASTE works by extracting text from test code, normalizing the text and applying detection rules to the normalized text [42].

SoCRATES was created by Bleser et al. [43] and uses static analysis to uncover test smells. In 164 projects written in Scala, they found that the tool achieved a precision of 98.94% and a recall of 89.59% [43]. The tool functions by retrieving syntactic information via an abstract syntax tree and semantic information, creating class hierarchies, finding test classes, connecting the test classes to production classes and, as a final step, detecting test smells [43].

Detecting if the test smell General Fixture exists, for example, was done by checking if every field in the fixture is used by the test case [43].

DARTS was introduced as an IDE plugin tool by Lambiase et al. [44]. DARTS uses the identification techniques for General Fixture and Eager Test that were proposed by Palomba et al. [42] and applies refactoring methods on the smelly tests to support development [44].

tsDetect is a tool created by Peruma et al. [29] and uses abstract syntax trees to find test smells in Android applications. The tool had an average precision of 96% and an average recall of 97% [29]. tsDetect generates an abstract syntax tree from a set of files containing unit tests and searches for patterns in the code based on defined rules [29].

RAIDE is a test smell detection and semi-automated refactoring tool introduced by Santana et al. [45]. RAIDE was developed as an IDE plugin for Java development and can indicate the lines where test smells appear [45]. It uses an abstract syntax tree to extract information from the test code and specialized algorithms to find and refactor the two test smells that the tool supports [45].

JNose Test was created by Virgínio et al. [46] as a test smell detection and metric collecting tool. It combines the test smell detection tool developed by Bavota et al. [31] and the code coverage metrics tool JaCoCo[1] [46]. The authors found, in their work, that test smells and test coverage are correlated [46]. Of the test smells included in Section 2.1.2, Assertion Roulette, Eager Test and Lazy Test were the smells with the highest correlation [46].

Table 3.1: Detection tool test smell implementations

	TestQ [39]	**Bavota et al. [31]**	**TestHound [40]**	**TestEvoHound [41]**	**TASTE [42]**	**SoCRATES [43]**	**DARTS [44]**	**tsDetect [29]**	**RAIDE [45]**	**JNose Test [46]**
Assertion Roulette	Yes	Yes				Yes		Yes	Yes	Yes
Conditional Test Logic								Yes		
Eager Test	Yes	Yes			Yes	Yes	Yes	Yes		Yes
For Testers Only	Yes	Yes								
General Fixture	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes		Yes
Indirect Testing	Yes	Yes								
Lazy Test		Yes				Yes		Yes		Yes
Mystery Guest	Yes	Yes				Yes		Yes		Yes
Resource Optimism		Yes[2]						Yes		Yes
Sensitive Equality	Yes	Yes				Yes		Yes		Yes
Sleepy Test								Yes		
Test Code Duplication	Yes	Yes								
Test Run War		Yes[3]								

Current test smell detection tools often make use of abstract syntax trees [43, 29, 45, 46]. However, they were only evaluated on open-source projects. This work expands on these previous methods by using syntax tree analysis to detect test smells on an industry codebase.

[1] https://www.eclemma.org/jacoco/

[2] Resource Optimism and Test Run War are grouped together with Mystery Guest

[3] See 2

3.2 Flaky test detection

There has been much work carried out in recent years devoted to detecting flaky tests [4]. This section covers some relevant detection methods for this work, where the most important methods are the rerunning strategy and using stressors to detect flaky tests.

3.2.1 Reruns

Rerunning the same test several times can detect flaky tests according to Lam et al. [47]. Since flaky tests pass and fail intermittently, the rerunning strategy should eventually lead to a different test outcome if the test is flaky. Lam et al. [47] ran test suites 4000 times each, leading to some flaky tests failing as infrequently as one time in all these runs. They state that this infrequency is one of the main challenges to this strategy as it takes substantial time and machine resources to find these flaky tests [47]. They also find that running test cases in isolation often changes the frequency of failures, which makes the process of debugging more difficult [47].

Bell et al. [48] evaluated three different re-running strategies in their work, analyzing 26 open-source Maven-based Java projects. The three strategies were the following: (1) if a test fails, run it up to five times until it passes and if it passes, it is deemed flaky, (2) start with strategy (1) and if the test has not passed, re-run it in a clean new JVM (Java Virtual Machine) up to five times and (3) start with strategy (2) and if the test has not passed, up to five times: clean the Maven project, reboot the machine and re-run the test [48]. Out of the 5,328 new test failures that were examined, 1,162 were found to be flaky by strategy (1), 4,186 were found to be flaky by strategy (2) and 5,075 were found to be flaky by strategy (3) [48]. Bell et al. [48] found that the first run in each of the three techniques were the most telling, meaning that running the test differently was more important than running it several times. The authors point out that this method can increase machine costs [48].

Google uses a strategy of rerunning tests up to 10 times when a test fails, to increase the likelihood of finding flaky tests [49]. They state that 2–16% of their computing resources goes towards re-running tests [49]. Microsoft uses an automated system that runs the test an extra time if it failed the first time and reports it automatically, if the test was found to be flaky [47]. Lam et al. [47] suggests that rerunning a failing test 5 times is sufficient to detect if a test is flaky with 87.8% accuracy.

3.2.2 Stressors

Silva et al. [5] studied 52 open-source projects to investigate how flaky tests were affected by resource limitations. They found that 46.5% of the flaky tests in these projects were affected by resource limitations and refer to these tests as RAFT (Resource-Affected Flaky Test) [5]. The authors also found that, out of the four tested resources, the most failure inducing resource was CPU availability; memory was the second most influential and disk and network were the least influential [5].

Silva et al. [6] developed a tool called Shaker that introduced noise in the execution environment. They used stress-ng, a tool that runs stressor processes that consume CPU and memory, to affect the running test [6]. The authors used a dataset of 11 open-source Android applications containing 75 flaky tests in total; 35 of the tests were randomly chosen for training and the remaining 40 were used to test and compare the rerunning strategy to Shaker [6]. 37.5% of the flaky tests were found with the rerunning strategy, while 95% were found using Shaker [6]. Shaker also managed to find 61 new flaky tests in the projects that had not been found by running the tests 50 times without added stress [6]. Since adding noise to the environment should increase the time it takes to run a test, this strategy is more expensive for the same number of reruns, but Silva et al. [6] show that less reruns are required to achieve good

results compared to the rerunning strategy. The work of Silva et al. [6] focuses on concurrency as the added stress was thought to affect the order in which events occur.

The Shaker tool was designed to reveal flakiness by running tests using different stress configurations [6]. The tests in the training dataset were run with randomly sampled configurations, and a set of configurations was selected based on their performance during these runs [6]. Shaker then used this limited set of configurations to run new tests with, a specified number of times [6]. Silva et al. [6] provided two alternatives for selecting a set of configurations: a greedy algorithm that selected the n configurations with the highest detection rate over the entire test set and a minimum hitting set (MHS) strategy that selected configurations that together have a detection rate over a threshold for every flaky test in the training set [6]. The authors found that there was no significant difference in the results of these two alternatives, but they were significantly better suited to detect flaky tests compared to randomly selecting configurations [6].

Similar to Shaker, Yost [7] developed a tool called StressSequence that reordered and executed tests with stressor processes. They used tests from 58 JavaScript projects and ran them 10 times with and 10 times without StressSequence [7]. They found that running stressor processes together with tests revealed more flaky tests related to the tests timing out and asynchronous operations such as updating UI [7].

3.2.3 Reordering

One of the most common causes of flaky tests is order dependency [1]. Zhang et al. [50] used ordering algorithms to detect tests suffering from this type of flakiness in 96 dependent Java tests from 5 issue tracking systems. They proposed 4 different algorithms to decide the order in which the test cases were run compared to the default order: reverse the order of the test cases, randomize the order of the test cases, run all k-permutations of the test cases and running a pruned set of the k-permutations of the test cases [50]. Lam et al. [51] continued the work of reordering tests by creating the tool *iDFlakies* for Maven-based Java projects. Compared to Zhang et al. [50], they randomly ordered the test classes and randomly ordered the test cases within them instead of interleaving test cases from different test classes [51]. Lam et al. [51] detected 422 flaky tests taken from 683 different projects and found that 50.5% of those flaky tests were order dependent. The strategies they used can be found in Table 3.2.

Table 3.2: Reordering strategies used by Lam et al. [51]

Strategy	Explanation
original-order	Using the original order of the test classes and test cases.
random-class	Using a randomized order of test classes but keeping the original order of test cases within the classes.
random-class-method	Using both a randomized order of test classes and a randomized order of test cases within the classes.
reverse-class	Using the reverse order of the test classes compared to the original order but keeping the original order of test cases within the classes.
reverse-class-method	Using the reverse order of the test classes and the reverse order of the test cases within the classes.

They found that *reverse-class-method*, as defined in Table 3.2, had the highest probability of finding any flaky test per run of the test suite; it had a probability of about 4 times higher than *random-class-method* and *reverse-class* [51]. They also found that *random-class-method* could uncover more flaky tests in total, as this introduced more randomness to the test run compared to the other strategies, increasing the probability of eventually finding a configuration that revealed the flakiness of the test [51].

We initially aimed to investigate test order dependency by reordering test cases. However, this strategy would have required its own experiment setup, and was abandoned early on in favor of our other experiments.

3.2.4 Non-deterministic specification

Some code constructs and methods are specified to be non-deterministic, such as iterating over a set or generating random values [52]. Tests making assumptions about deterministic behavior when there are no guarantees can be considered as flaky [52]. Shi et al. [52] calls this type of flaky behavior for ADINS (Assumes a Deterministic Implementation of a Non-deterministic Specification). Gyori et al. [53] created a tool called NonDex that modifies non-deterministic APIs for the Java standard library so that they explore different orders and responses. If the test passed for one ordering but not another, it meant that the test was flaky [53]. This tool found 21 new bugs and 54 known bugs in 8 open-source Java projects [53].

Some of the flaky tests found in our results were closely related to this flakiness root cause. However, the tool presented by Gyori et al. [53] would not be applicable for the industry codebase as the non-determinism presented in Section 5.2 did not stem from standard library specifications.

3.2.5 Static code analysis

Waterloo et al. [54] conducted a study where they predicted flakiness by using static code analysis on 12 Java projects. They generated abstract syntax trees from the test code and detected problematic patterns from the syntax tree. The problematic patterns were divided into three categories: *Inter-Test Dependencies Patterns*, *External Dependencies Patterns* and *Intra-Test Patterns*. Each category consisted of three patterns, see Table 3.3.

Table 3.3: Problematic patterns in test code [54]

Inter-Test Dependencies Patterns	Explanation
Test Calls Test	A test calls another test and affect each other by writing to a static field.
Shared Static Field	One test writes to a static field and another test reads the static field causing dependence.
Shared Stream	Two tests share a resource implemented as a stream.
External Dependencies Patterns	**Explanation**
Timing Dependency	The test code tries to wait for the asynchronous event to finish by resorting to methods that are not directly linked to when the event finishes.
System Dependency	The test writes to or reads from the system state, making assumptions about the machine.
Network Dependency	The test makes assumptions about the network connection.
Intra-Test Patterns	**Explanation**
Over Checked	The test checks more values in a returned object than necessary for the test.
Vague Exception	The test code tries to catch an exception and uses a general exception type instead of a specific one.
Under Checked	The test insufficiently checks the results of the system under test (SUT).

Waterloo et al. [54] came to the conclusion that the most prominent patterns were, starting with the most prevalent: Timing Dependency, Under Checked and Vague Exception. Timing

Dependency could be found without any false positives in the 31 cases that were analyzed by the authors [54]. System Dependency, Vague Exception and Under Checked all had false positive rates lower than 25% [54].

A similar method to detect flaky tests using static code analysis, as Waterloo et al. [54] performed, is used in this work. Our method focuses mostly on anti-patterns in tests without external influence such as Intra-Test Patterns, although using another vocabulary to describe these test smells.

Summary: This chapter reviewed relevant studies on software testing and test flakiness. It introduced relevant test smell detection tools developed over the years and explored various techniques for flaky test detection, with a focus on rerunning tests and using stressors to induce flakiness. The next chapter, Method, details the methodology of the thesis.

---

4 Method

This chapter explains the methodology used for answering the three research questions. In Section 4.1, we present our experiment of mass test execution to detect resource-affected flakiness. In Section 4.2, we present how we analyzed test logs and flakiness-related commits to identify flakiness root causes. Finally, in Section 4.3, we present our source code analysis to find potential flakiness predictors in test smells and code metrics.

4.1 Computational resources and test flakiness

RQ1: How does computational resource availability correlate with test flakiness?

To answer RQ1, we conducted an experiment of executing test cases under simulated CPU usage. In this section we describe our dataset of test cases (Section 4.1.1), test environment (Section 4.1.2), test execution framework (Section 4.1.3), experiment (Section 4.1.4) and data analysis (Section 4.1.5).

4.1.1 Dataset

The dataset investigated for RQ1 consisted of 199 test cases selected from an industrial C++ project. The project contained a large set of test cases on multiple levels, so we included only a filtered subset of the most relevant test suites.

Firstly, we included test cases only from test suites with recent flaky executions. Since our primary goal was to find a relation between flakiness and CPU usage, and not specifically to identify new flaky tests, we argued that targeting known flaky test suites would produce more data with fewer required executions. Furthermore, we focused specifically on functional integration test suites, excluding structural tests. Structural tests, which depend primarily on source code structure, were not considered relevant since known flakiness root causes (e.g., Async Wait, Concurrency; Section 2.2.1) typically manifest in functional tests. We selected integration tests because they most often appeared to fail for reasons aligned with test flakiness root causes documented in literature (e.g., Async Wait, Concurrency). Although we also considered unit tests, few were identified as flaky, so they were excluded. System-level tests were not considered. After filtering, we ended up with a total of 199 test cases

from 12 functional integration test suites. Each test suite was included as a whole, without excluding specific test cases.

4.1.2 Test environment

The test environment used for executing the tests, both by developers and for our experiment, was asynchronous in nature and therefore required many asynchronous operations in the test cases. Figure 4.1 illustrates the abstract relation between a test suite and the system under test (SUT), where a test suite could only influence the SUT through its public API and then validate the set of signals sent from SUT to test suite. Because a test case could not immediately observe or validate SUT behaviors, the test cases had to utilize asynchronous assertions. For instance, Listing 4.1 describes a basic test case. In the test case, a SUT behavior is triggered through the function call to *triggerBehavior1*, upon which an assertion (expectation) is added. The synchronization operation puts the test case in a waiting state, where it would not wake before all expected signals had been validated.

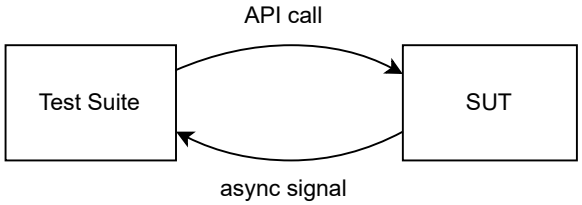


Figure 4.1: Test environment, highly abstracted

```
1 TEST(TestCase_1) {
2     triggerBehavior1();
3     addExpectation1();
4     synchronize();
5 }
```

Listing 4.1: Example basic test case

In comparison, Listing 4.2 illustrates how the test could have been written without the limitations of the asynchronous test environment. Here, *triggerBehavior1* is a synchronous operation that does not complete until the behavior is finished, so that synchronization directives and asynchronous assertions are unnecessary. Furthermore, Listing 4.3 describes a more advanced test case, consisting of multiple synchronization operations. The use of multiple synchronizations marks the test as a multi-phase test, where multiple behaviors are triggered and validated in sequence. Furthermore, the test case performs a configuration operation in the second phase (line 13), where the SUT is interacted with before the tested behavior is triggered. The test case then waits for the configuration to activate, but because of limited observability, the test can only wait for a fixed amount of time and hope that it is enough. While some observability and asynchronicity issues likely could have been solved by adding more signals in the production code, it is bad testing practice to add production logic only intended for testing [13, 30].

```
1 TEST(TestCase_1) {
2     triggerBehavior1();
3     assert(expectation1);
4 }
```

Listing 4.2: Example basic test case, without test environment limitations

```
TEST(TestCase_2) {
    triggerBehavior1();
    synchronize();

    configureBehavior2();
    sleep(1);
    triggerBehavior2();
    addExpectation2();
    synchronize();
}
```

Listing 4.3: Example advanced test case

The test environment was executed on a high performance server machine dedicated to the experiment. No other users or processes could access the server, allowing the tests to run in isolation and under full control.

4.1.3 Test execution framework

A custom test execution framework was constructed to control the test environment during the experiment, illustrated by Figure 4.2. The framework was tasked with initiating the test environment with selected test suites and simulating CPU usage on the underlying hardware. Furthermore, it monitored system resource utilization and documented both the outcomes of the test cases and the resource usage over the lifetime of each test case, in order to produce our results.

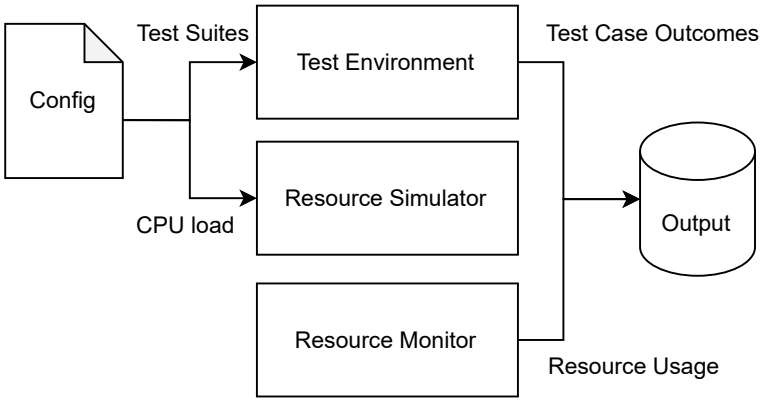


Figure 4.2: Test execution framework

The framework acted as a driver for the test environment, by compiling and running tests over long periods of time. Because the build step was very slow, we enabled the framework to reuse the same build artifact for many test executions. This severely reduced the time needed for execution, as the original test framework compiled the test environment for every execution. Furthermore, the framework was built to run multiple tests in parallel to reduce the execution time. This had the added benefit of making the system resource usage of the server more realistic, as a production test server would likely never run just one test suite at a time. We argued that a more realistic resource usage would both provide recognizable results, in line with flakiness already detected by subject matter experts, and make our results more transferrable to real world scenarios. The impact of parallel execution was later verified to be minimal on test case outcomes.

Each test case in the dataset was executed sequentially within its test suite, and each suite sequentially relative to the other suites. The framework implemented an early abortion rule, which mandated the termination of all test suites upon the failure of the first test case. This

rule was a feature of the test environment, and we decided to maintain it to align with real-world scenarios. We argued that, while early abortion might reduce the number of samples of later test cases, it would not affect the pass/fail ratio of individual test cases, under the assumption of test order independence. Since the environment rebooted for every test case for a clean start, and no test interacted with entities outside of the environment, we deemed the risk of test order dependency to be extremely low.

Resource simulation was performed using stress-ng [1], an open-source tool for stress testing systems with simulated usage of various resources. Early on we selected to only investigate CPU usage, because early trials showed large impact of CPU usage, but no apparent effect of neither memory or disk usage.

Resource monitoring was implemented using operating system specific utilities (vmstat[2]) that could monitor CPU, memory and disk usage.

4.1.4 Experiment

The experiment was performed by executing the test suites of the dataset thousands of times under various simulated CPU usages. In the initial phase, we tried out different configurations of parallelism and CPU usage to quickly increase the number of documented flaky failures and identify which test suites that exhibited flakiness. Although the test suite dataset had been selected due to known flakiness or recent flaky failures, not all test suites manifested flakiness. Once we concluded that we could not obtain any failures from some, we excluded them from the experiment to reduce time required for running the experiment.

The flaky test suites were executed 23,000 times. Although not every test case was executed the same number of times due to the test environment early abortion rule, no flaky test was executed less than 20,000 times. Furthermore, all test cases were executed at least 3000 times per CPU usage interval of 0–20%, 20–40%, 40–60%, 60–80% and 80–100%.

4.1.5 Data analysis

The output of the experiment, i.e., test case outcomes mapped to CPU usage, was visually inspected through plots to identify relations between CPU usage and failure rates of individual test cases. Each test case was given a unique identifier. We found three distinct patterns, and labelled each test case accordingly: non-flaky test, resource-affected flaky test (RAFT) and resource-independent flaky test (RIFT). Figure 4.3 shows an examples of the visual representation used for inspecting test case 122, which was clearly a resource-affected flaky test.

[1] https://github.com/ColinIanKing/stress-ng

[2] https://www.ibm.com/docs/tr/aix/7.1?topic=v-vmstat-command

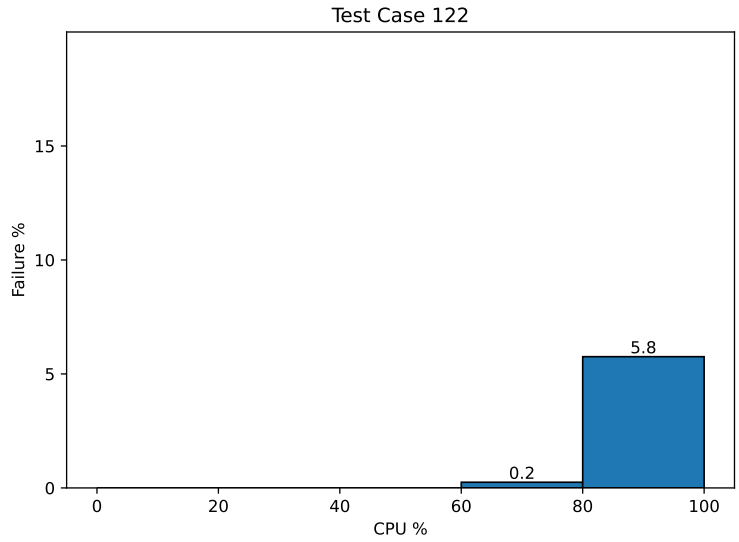


Figure 4.3: Example bar plot, showing failure rate of Test Case 122 over CPU usage

During inspection, we found some test cases to fail a very small percentage of times. Therefore, we treated any test case as non-flaky if it had a failure rate of less than 0.1% for all 20% CPU usage intervals shown in Figure 4.3.

To verify that running the tests in parallel did not affect the experiment outcome, we checked that failures were similarly distributed across CPU usage independently from parallelism. We selected 1, 15 and 30 as target degrees of parallelism for this check. Running 30 tests in parallel drastically increased the number of tests we could execute per hour. Conversely, running one test at a time limited our data collection. To obtain enough data for the main research question, we ran only 3,000 executions without parallelism, while the higher degrees of parallelism were used for over 10,000 executions each. The impact of parallelism on the results was visualized and qualitatively analyzed.

Once positive relations between CPU usage and failure rate were established, we performed statistical testing using a one-sided Wilcoxon rank-sum test [55] for every flaky test case to validate the assigned labels. We used the following model:

Samples:

$$X = \{x_1, x_2, \ldots, x_{n_X}\} \quad \text{(CPU values for failed tests)}$$
$$Y = \{y_1, y_2, \ldots, y_{n_Y}\} \quad \text{(CPU values for passed tests)}$$

Hypotheses:

H_0 : The distributions of X and Y are identical

H_1 : The distribution of X is stochastically greater than the distribution of Y

Test Statistic:

$$W = \sum_{x \in X} R(x) \quad \text{(Sum of ranks for observations in } X\text{)}.$$

Under H_0, the test statistic W is approximately normally distributed [55]:

$$W \sim N\left(n_X \cdot \frac{n_X + n_Y + 1}{2}, \sqrt{\frac{n_X n_Y (n_X + n_Y + 1)}{12}}\right).$$

Therefore, we could simply obtain p-values from normal distribution tables. We used a significance level of 0.001 to reject H_0 if $p < 0.001$ and prove that the CPU usage of failed tests was higher than that of passed tests, making the test case a RAFT.

We chose the Wilcoxon rank-sum test because it is non-parametric, meaning that no assumption of normally distributed samples was required [55]. Our samples, in the form of CPU usage values for passed and failed tests, could not be assumed to be normally distributed. Furthermore, we found no central tendency in the samples, making them unlikely to be normally distributed. Using parametric tests such as the t-test, which require approximately normally distributed samples [55], would therefore not have been appropriate.

Because we had executed more tests with higher CPU usage than lower, we sampled the executions of each test case equally by dividing the executions into CPU usage buckets of 20% intervals and sampling an equal number of executions from each bucket. This provided a balanced set of executions, unaffected by how we had configured them.

4.2 Test flakiness root causes

RQ2: How do specific root causes of flaky tests and computational resource availability contribute to flakiness?

Two approaches were used to answer RQ2: analyzing test logs from the experiments described in Section 4.1 and collecting fixes of flaky tests from the version control system. These approaches are described in detail in Section 4.2.1 and Section 4.2.2 respectively.

4.2.1 Analysis of test logs

The experiments described in Section 4.1 produced logs as an additional output. These logs were generated automatically while running tests and contained information about the test runs. The test logs were filtered and sorted by output similarity. The outputs were classified according to which root cause was the source of the failure. The filtering and classification are described in this section.

The log files generated by the test driver included important information about the failures such as which test cases passed and failed, which expectations were not met and the state of the system at the time of the failure. The state of the system included relevant context about the events that had occurred during the execution.

The test cases that had a failure rate of less than 0.1% were filtered out from further analysis to only investigate the most flaky test cases. This threshold was chosen to focus on the most frequently occurring issues, optimizing the use of time and resources. The passing runs of the flaky test cases were also filtered out, since they could not provide information about potential root causes, leaving only the failing test runs for the flaky test cases.

Using the logs produced by the flaky tests, the test cases were grouped by root cause. The classification of root causes was done by a subject matter expert using the root causes listed in Table 4.1. The expert could provide information about the exact root cause based on the logs, previous bug reports and source code analysis. They had more than 20 years of experience in the industry and more than 5 years of experience in the project. In theory, one test case could fail because of several different root causes. However, the subject matter expert concluded, based on previous data and the test logs presented, that each of the flaky tests only suffered from one root cause each. Therefore, only one root cause was assigned per test case in this study. The grouping provided a basis to investigate if certain root causes consistently had a higher failure rate in environments with high CPU usage compared to environments with low CPU usage.

4.2.2 Fixes of flaky tests

The second method used to answer RQ2 is presented in this section. Fixes of flaky tests were gathered and categorized and the fixed test cases were run before and after the fix was implemented. Figure 4.4 describes an instance of this process, where Commit #1 represents the last commit where the test case was flaky and Commit #2 represents the implemented fix.

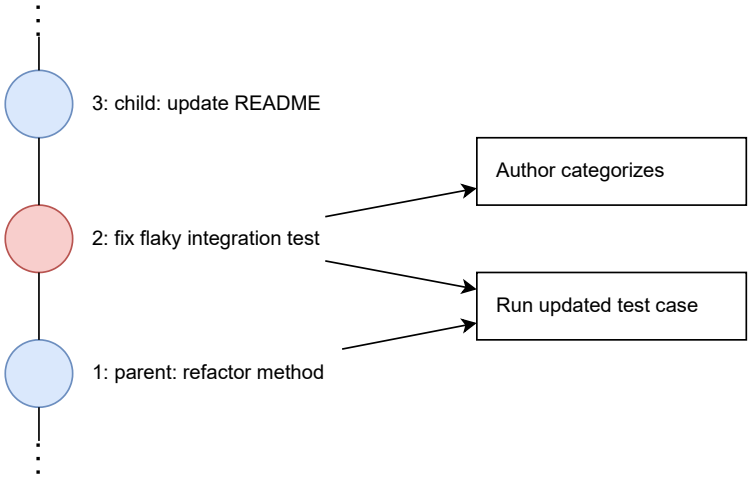


Figure 4.4: Overview of the method using fixes of flaky tests

Commits that aimed to fix flaky tests were collected. This was done by searching for commit message patterns in the version control system that implied this type of fix. Commits from the past 12 months were found by searching for the following words in the version control history: “flaky”, “flake”, “intermit”, “flaki”, “fragile” and “brittle”. 63 commits following this description were found. As shown in Figure 4.4, Commit #2 would have been found using this method since the commit name contained the word “flaky”.

To categorize the root causes that caused the test to exhibit flaky failures, a survey was sent to the individual authors of the commits where the fixes were introduced. The survey contained a link to the author’s change and a list of root causes formulated as shown in Table 4.1, using the explanations given by Parry et al. [4]. These root causes were chosen because they had appeared in an internal survey at the company and were therefore thought to be relevant to investigate. The authors were asked to identify which root cause was linked to each of the flaky tests that they had fixed. 62 of the 63 commits were classified with a root cause via this survey. The final commit was reviewed, but it could not be concluded which root cause was the most appropriate. The root causes with less than 10 instances were filtered out to focus the experiments on the root causes with the most data.

Table 4.1: Flakiness root causes definitions from author survey

Common root causes	Explanation
Concurrency	"Test that invokes multiple threads interacting in an unsafe or unanticipated manner. Flakiness is caused by, for example, race conditions resulting from implicit assumptions about the ordering of execution, leading to deadlocks in certain test runs" [4]
Test Order Dependency	"Test that depends on some shared value or resource that is modified by another test that impacts its outcome. In the case where the test run order is changed, these flaky tests may produce inconsistent outcomes, since the dependencies upon tests previously executed beforehand are broken" [4]
Async Wait	"Test makes an asynchronous call and does not explicitly wait for it to finish before evaluating assertions, typically using a fixed time delay instead. Results may be inconsistent in executions where the asynchronous call takes longer than the specified time to finish, leading to the flakiness" [4]
Unordered Collections	"Test assumes a particular iteration order for an unordered collection type object. Since no particular order is specified for such objects, tests that assume they will iterate in some fixed order will likely be flaky due to a variety of reasons, e.g., implementation of the collection class" [4]
I/O	"Test that is flaky due to its handling of input and output operations. For example, a test that fails when a disk has no free space or becomes full during file writing" [4]
Time	"Test relies on local system time and it may be flaky due to discrepancies in precision and timezone, e.g., failing when midnight changes in the UTC timezone" [4]
Test Case Timeout	"Test specified with an upper limit on their execution time. Since itś usually not possible to precisely estimate how long a test will take to run, by specifying an upper time limit a developer may run the risk of creating a flaky test, since it could fail on certain executions that are slower than anticipated" [4]
Test Suite Timeout	"Test is part of a test suite with a limited execution time. Intermittently fails, because it happens to be running once the test suite hits an upper time limit" [4]
Other	(please describe the root cause in general)

Experiments

Each of the fixed tests were run 2400 times before and after the commit that fixed the flaky test. In Figure 4.4, the updated test case is run for both Commit #1 and Commit #2. CPU stress was added to the system while the tests were running in the same manner as for RQ1, see Section 4.1. Previous experiments had shown that CPU usage under 80% did not have a large impact on the failure rate of RAFTs. Therefore, the CPU loads that were used were 0, 20, 40 and 80%, with 400, 400, 400 and 1200 executions respectively. This was done to compare compare high CPU usage with a distribution of lower CPU usage to find significant differences in failure rates. A minor subset of executions with CPU load of 40% or lower had a CPU usage exceeding 80%. Since these instances were so few, they were excluded to simplify the post processing of the executions.

Varying CPU loads were used to uncover flaky behavior patterns and better reflect real-world scenarios. 60% was not chosen since the CPU usage had been observed to exceed 80% when running many executions in parallel. Changing CPU load during executions took time, so to save time only three CPU loads lower than 80% were chosen.

Compared to RQ1, only the test cases mentioned in the commit or internal task were run. This was done to increase the number of executions that was possible. It increased the throughput for two reasons: test cases take time to run so decreasing the number of test cases that were run decreased the total time for each execution, and if a test case failed before the test case that was fixed, then the test case that was fixed would not run and the execution would not be used in the final results.

Data analysis

To verify that running only one test case at a time did not affect the experiment outcome, one commit was run using the entire test suite as an additional experiment. This configuration was executed as many times as the single test cases. The distribution of failure rates was analyzed and compared to only running the updated test case.

Each test case that was affected by one of the chosen commits was analyzed by plotting and visually inspecting to find differences in failure rates for CPU usage lower than 80% compared to CPU usage equal to or higher than 80%. The test cases were labelled as RAFT or RIFT based on how the CPU usage affected the results. This was done to investigate if the flakiness of the test cases was resolved, improved, or increased after the fix.

4.3 Test smells and code metrics

RQ3: How do test smells, code metrics and computational resource availability predict test flakiness?

To answer RQ3, we expanded upon on the data collected for RQ1 by extracting test smells and code metrics using a custom parser tool. Smells, metrics and previous flakiness data was qualitatively analyzed to find potential predictors of resource-affected flakiness. In this section we present the dataset (Section 4.3.1), selected test smells (Section 4.3.2), selected code metrics (Section 4.3.3), parser tool (Section 4.3.4), metrics computation (Section 4.3.5), test smell detection (Section 4.3.6) and data analysis (Section 4.3.7).

4.3.1 Dataset

The dataset used for answering RQ3 was derived from two sources: the output of the experiments performed for RQ1, and the source code of the analyzed test cases. From RQ1, we reused the flakiness labels assigned to the test cases, i.e., 188 non-flaky tests, 8 RAFTs and 3 RIFTs. The second part of the dataset consisted of the source code of each labelled test case. The code was written in the C++ programming language and used the Google Test framework. Specifically, the test source code largely consisted of asynchronous operations due to the nature of the test environment described in Section 4.1.2. The test source code was used to extract test smells and code metrics, as described by Sections 4.3.5 and 4.3.6, for comparison with the flakiness labels.

4.3.2 Selecting test smells

Initially, 13 test smells were selected for investigating RQ3. However, during implementation of the test smell detection rule, eight of these were excluded due to project-specific reasons. Table 4.2 shows the included and excluded smells. In this subsection, we present each test smell and motivation to include or exclude it. Notably, all presented smells besides Sleepy Test and Conditional test Logic are part of the original set of test smells by Van Deursen et al. [13]. Since these original smells appear frequently in literature, we considered them relevant to investigate.

Table 4.2: Test smells selected for RQ3

Included	Sleepy Test
	Conditional Test Logic
	Eager Test
	Lazy Test
	Test Code Duplication
Excluded	Mystery Guest
	Resource Optimism
	Test Run War
	General Fixture
	Assertion Roulette
	Indirect Testing
	For Testers Only
	Sensitive Equality

Sleepy Test was included because previous research has linked it to test flakiness [1, 11] and because it is easy to detect. The use of thread sleeps in test code may imply synchronization issues with external resources (i.e., Async Wait) or the use of multithreading in the test code (Concurrency), both of which may cause flakiness [1]. The detection of Sleepy Test is only a matter of finding calls to standard sleep methods, and their project specific equivalences. Furthermore, we argued together with subject matter experts that the presence of constant-time thread sleeps could specifically cause resource-affected flakiness. Increased resource usage may delay external processes, sometimes making the selected thread sleep duration insufficient for waiting on said processes. Due to the asynchronous nature of the test environment (Section 4.1.2), it was believed that the usage of thread sleeps would be impactful on predicting flakiness.

Conditional Test Logic was included because it may cause both obscurity to the reader and flaky behavior [30], and because it is easy to detect. Conditional logic creates alternative execution paths, which forces the reader to decode the conditional logic in order to understand what the test actually executes. If the conditions are non-deterministic with regards to the test input, then the test execution is also non-deterministic, which may cause flakiness. Detecting conditional logic is easy because it is manifested in explicit control statements such as if, $while$ and for.

Eager Test, **Lazy Test** and **Test Code Duplication** were included as the presence of either one may imply maintainability issues. Eager tests exercise multiple behaviors per test while lazy tests exercise the same behavior over multiple tests, both of which goes against the principle of one test case per behavior. Changing a production behavior related to such tests becomes difficult as it is not obvious which tests that need to be changed, which may cause mistakes and problems in the long term. Duplicated test code causes similar problems as it requires maintaining consistency between all duplicated code instances when making changes. We argued that the maintainability issues caused by these smells increase the risk of developer mistakes, in line with Bavota et al. [31], and that they consequently increase the risk of flakiness being introduced.

Mystery Guest, **Resource Optimism**, **Test Run War** are test smells defined by unsafe or obscure usage of external resources [13]. Tests exhibiting these smells may randomly fail if the resource in question becomes unavailable for any reason. We initially argued that these smells could directly impact flakiness, but when analyzing the source code to find patterns identifying the smells, we found no usage of external resources. When asking the subject matter experts, they explained that this was intentional, as they were already aware of the potential issue with external resources and prohibited it in testing. The smells were therefore impossible to be introduced by developers, forcing us to exclude them from the investigation. Similarly, **Assertion Roulette** was non-existent in the codebase as the only allowed assertion

operations enforced descriptions and source code location. Assertion Roulette could have otherwise been used to indicate maintainability issues, and therefore error-prone tests.

General Fixture refers to a test fixture being used only partially by some test cases, causing incoherency in the fixture code. We initially included the smell for the same reason as other maintenance-related smells, i.e., higher risk of mistakes. However, measuring this during code analysis would have required identifying and mapping out the contents of the fixture classes, and how they were utilized by the test cases. Such a feature would have added substantial complexity to the parser, magnitudes higher than for other smells. Excluding this smell allowed us to better focus our efforts on other parts of the investigation.

Indirect Testing and **For Testers Only** both depend on the source code of the SUT, and both cause maintenance issues. However, in part due the fact that the tests and SUT were compiled and executed as separate programs, we could not link test code with SUT code to analyze them together. Therefore we chose to ignore all production code, and these smells.

Sensitive Equality can cause unexpected test failures when a string representation is modified while it is being used for equality checks in test code. We did not not find it in a single test case, and therefore excluded the smell. Unlike other non-existent smells, such as Assertion Roulette, there was no explicit rule or check against this smell. Instead, it appeared that the developers knew well to avoid using string representations for equality.

4.3.3 Selecting code metrics

Test case code metrics were selected to give an abstracted insight into the structure and behavior of a test case. The following code metrics were investigated, and are described in this section: *Lines of Code*, *Cyclomatic Complexity*, *Sleeps*, *Assertions* and *Synchronizations*.

Lines of Code (LOC) measures how many lines of source code that a test case consists of, with a higher LOC value indicating a larger test case. **Cyclomatic Complexity (CC)** similarly measures the structural complexity of a program (test case) by considering control flow. It is formally defined as $CC = 1 + M - N$, where M and N are the number of edges and nodes respectively in the control flow graph of a program [56]. A higher CC indicates a higher number of logical paths, which can make a program harder to understand. A CC value of 1 means that a program has a single sequential logical path. Both LOC and CC were selected as they may point to maintainability issues in different ways, and therefore indicate error-prone test cases. The metrics were also easily computed.

The **Sleeps** metric measured the number of constant-time thread sleeps in a test case, and was included for the same reason as the Sleepy Test smell, i.e., because constant-time thread sleeps may cause resource affected flakiness. Because of the asynchronous test environment and limited observability (Section 4.1.2), constant-time thread sleeps were known to exist in the test code.

The **Asserts** metric measured the number of (asynchronous) test assertions in a test case, while the **Synchronizations** measured the number of synchronization points (Section 4.1.2). Discussions with subject matter experts revealed that mistakes involving these operations could cause flakiness. Furthermore, we argued that the metrics also could be used to indicate complicated test logic, similarly or alternatively to Cyclomatic Complexity. Therefore, we included these metrics in the investigation.

4.3.4 Test source code parser

A test source code parser tool was created to convert raw test source code into abstracted formats usable for computing code metrics and detecting test smells. To achieve this, we enabled the tool to extract abstract code representations as syntax trees and lists of operations, described in this section.

Abstract syntax tree representation

The abstract syntax tree (AST) representation of the test code, illustrated by Figure 4.5, enabled structured source code analysis while abstracting C++ language details. Listing 4.4 shows an example test case, utilizing a fixture to send a message as an administrator and validate the response. Using Clang [3], the AST shown in Figure 4.5 could be extracted from the test case, producing an abstracted and structured model of the source code that could be traversed recursively.

```
1  // file: test.cc
2  #include "fixture.h"
3  #include "gtest.h" // Google Test
4
5  TEST_F(Fixture, MyTestCase) {
6      role = "admin";
7      send("message");
8      expectOk();
9  }
10
11 // file: fixture.cc
12 void Fixture::send(string s) {
13     trace(s)
14     if (role == "admin")
15         bus.sendAsAdmin(s);
16     else
17         bus.send(s);
18     expectReceived();
19 }
```

Listing 4.4: Example Test Case, using Google Test framework

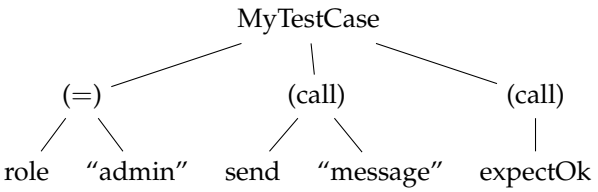


Figure 4.5: Simplified AST representation of MyTestCase in Listing 4.4

Furthermore, each node in the AST was annotated with detailed information, such as type, scope and symbolic reference links. For example, the call to *send* on line 7 in Listing 4.4 could be traced the function declaration of *Fixture::send* in another file (line 12 in the Listing 4.4), providing a way to extract fully-qualified names for accurate comparisons. Tracing references to declarations across files was critical for extracting metrics and test smells, and resolving such C++ symbolic references from the raw source code text without a competent tool such as Clang would have been beyond our abilities.

Test case list representation

While the AST code representation helped analyze test cases one by one, enabling code metric computation and detection of some test smells, a few test smells (i.e., Lazy Test and Test Code Duplication) required pairwise test case comparison. It was difficult to utilize the tree-based structures for this, which led us to enable the parser tool to also extract the lists of operations from test cases, illustrated by Listing 4.5. We defined a test case operation as either a function call or an assignment to a fixture value. In both cases, names were expanded to their fully

[3] https://github.com/llvm/llvm-project/tree/main/clang/bindings/python

qualified equivalences, and operands were stripped. Other operations, such as assignments to local variables, were ignored to allow more matches when comparing operations between tests. Operands were stripped for the same reason. We argued that details beyond function calls and fixture mutation were rarely relevant when comparing tests, and would therefore lead to worse results if included.

```
1 Fixture.role =
2 Fixture::send
3 Fixture::expectOk
```

Listing 4.5: List reperesentation of MyTestCase in 4.4

For some metrics and smells we needed not only the operations directly related to a test case, but also the operations of helper functions used by it. For example, in Listing 4.4 we see that the function *Fixture::send* calls *expectReceived*, which could be useful to capture when comparing lists of operations with each other. Using the already generated AST of Fixture::send, illustrated by Figure 4.6, the previously shown *direct operations* could be expanded to *effective operations* shown in Listing 4.6. As we can see, the structure of the operations is lost, making it seem like *MessageBus::sendAsAdmin* and *MessageBus::send* are called sequentially when they actually belong to different if-branches. This was an accepted tradeoff, as representing both the set of operations and code structure in a format usable for comparison performed poorly. We tried multiple tree edit distance algorithms to use AST representations for comparison, but none performed even close to simply comparing operation lists. We also note that the call to *trace* was not included in the effective operation list. This was because some operations were explicitly ignored as they did not meaningfully impact the test behavior, which was decided after discussion with subject matter experts. We found that a vast majority of sleeps, asserts and synchronizations were performed by helper functions, in our dataset of test cases. By not considering effective operations, this information would have been missed.

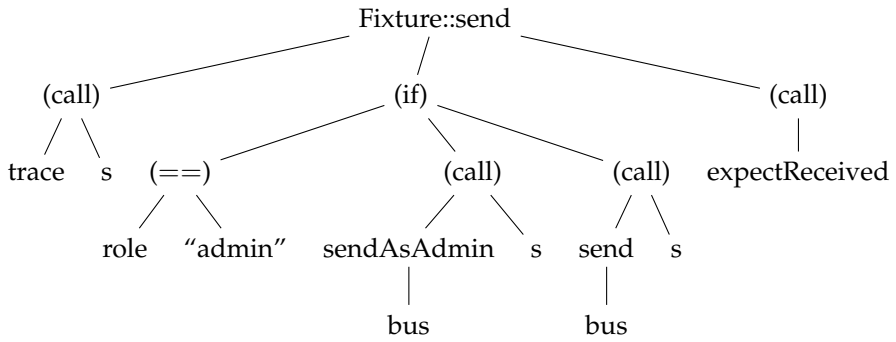


Figure 4.6: Simplified AST representation of Fixture::send in Listing 4.4

```
1 Fixture.role =
2 MessageBus::sendAsAdmin
3 MessageBus::send
4 Fixture::expectReceived
5 Fixture::expectOk
```

Listing 4.6: Effective list reperesentation of MyTestCase in 4.4

4.3.5 Computing code metrics

The selected code metrics were computed automatically using both the AST code representation and list code representations acquired from the custom parser. Lines of Code was obtained directly from Clang, the underlying parsing tool.

Cyclomatic Complexity

Cyclomatic Complexity was computed by counting the number of decision points in the code AST, such as if-statements and for-loops. Every decision point in a program creates two execution paths: one for when the decision is taken and one for when it is not. Since Cyclomatic Complexity is calculated as $CC = 1 + M - N$ where M is the number of edges and N is the number of nodes in the program control flow graph [56], and each decision point adds two edges and one node, we could simplify the formula to $CC = 1 + K$ for K decision points.

The following nodes types were considered decision points when computing Cyclomatic Complexity: if-statement, for-loop, for-range-loop, while-loop, do-while-loop, switch-statement, case-statement and the ternary operator. Short-circuiting logical operators (&& and ||) were excluded because, in the analyzed source code, their usage did not involve right-operand function calls. Such instances, where the right operand of a short-circuiting operator involves a function call, would be decision points due to the conditional evaluation of the function. However, in the source code, short-circuiting operators were used only as basic boolean operators, which do not increase Cyclomatic Complexity. The default-statement, related to switch-statement, was also explicitly excluded. Adding a default-statement creates a new control flow path, but also removes the implicit path that performs neither switch-case and instead jumps to the end of the switch-statement. Therefore, the net effect of a default-statement on Cyclomatic Complexity is zero.

Sleeps, asserts and synchronizations

The number of sleeps, asserts and synchronizations were counted by searching the AST of a test case for calls to certain domain specific functions, identified by subject matter experts.

Effective metrics

Each metric was computed in two ways: a *direct metric* and an *effective metric*. The direct metric was based only on the code representation of the test case itself, while the effective metric was aggregated over the test case and all relevant testing helper functions called by the test case. For instance, effective LOC was measured as the sum of lines of code of a test case and each relevant testing helper function it used. A helper function was considered a testing helper function if it performed test assertions or in other ways behaved similar to a test case. A testing helper function was classified as irrelevant if it was explicitly flagged so by the subject matter experts, which was done for a subset because of domain specific reasons.

By including the effective code metrics, we could use a code representation more aligned with how a developer views the code. For maintainability metrics such as LOC and CC, we argued that maintainability in testing helper functions may be as important as directly in the test function if a developer has to maintain both functions equally. For behavioral metrics such as Sleeps we argued that a thread sleep may be equally relevant if it is performed by a helper function as if it was performed by the test function itself. It was also concluded by subject matter experts that the implementation of some relevant testing helper functions had a major impact on flakiness in the investigated project.

Computing effective Cyclomatic Complexity required special care, as the sum of multiple functions' Cyclomatic Complexity scales the constant 1-term with the number of functions even if none of those functions have any decision points. Therefore, the constant 1-term was

subtracted from the formula, giving

$$CC_{\text{effective}} = 1 + \sum(CC - 1) = 1 + K_{\text{effective}}, \tag{4.1}$$

where $K_{\text{effective}}$ is the number of decision points in the test case and its helper functions.

4.3.6 Detecting test smells

The selected test smells were automatically detected based on test case code metrics described in Section 4.3.5 and test case operation lists generated by the custom parser described in Section 4.3.4.

Sleepy Test and Conditional Test Logic

The Sleepy Test and Conditional Test Logic smells were trivially derived from the effective metrics Sleeps and Cyclomatic Complexity:

$$\text{Sleepy Test} = \begin{cases} \text{True}, & \text{if Sleeps} > 0 \\ \text{False}, & \text{otherwise}, \end{cases} \tag{4.2}$$

$$\text{Conditional Test Logic} = \begin{cases} \text{True}, & \text{if Cyclomatic Complexity} > 1 \\ \text{False}, & \text{otherwise.} \end{cases} \tag{4.3}$$

We considered it appropriate to use effective metrics over direct metrics when detecting these smells. For Sleepy Test, we argued that a thread sleep is equally relevant if it is performed directly by the test as by a helper function, since the functional outcome of the thread sleep is the same. Likewise, we argued that Conditional Test Logic should have a similar impact on flakiness if it occurs directly in the test code as if it occurs in a helper function.

Eager Test

The Eager Test smell was similarly detected using the Synchronizations metric. Due to the asynchronous nature of the test environment (Section 4.1.2), the source code a of a test case was sometimes divided into multiple test phases, each phase ended by a synchronization directive that ensured successful completion of the phase. During each such phase, a different behavior of the SUT was typically exercised. Therefore, the presence of multiple such synchronizations in a single test case indicated multiple tested behaviors. The rule was therefore defined as

$$\text{Eager Test} = \begin{cases} \text{True}, & \text{if Synchronizations} > 1 \\ \text{False}, & \text{otherwise.} \end{cases} \tag{4.4}$$

Attempts were made to find alternative code patterns that could indicate Eager Test. In literature, Eager Test is defined as a test method that calls multiple production methods [13, 11], which refers to JUnit concepts that are not directly transferrable to the investigated C++ project. In the investigated project, the SUT was truly a black box, making it difficult to understand which behaviors were triggered by a test case. We were unsuccessful in deriving a model of how test cases implicitly interacted with specific production behaviors, that could have otherwise been used instead of the more heuristic rule of multiple synchronizations.

Lazy Test

The Lazy Test smell was detected by comparing the lists of effective operations, extracted by the custom parser (Section 4.3.4), between pairs of test cases. Literature defines the Lazy Test smell as multiple tests exercising the same behavior, possibly with different validation

criteria [13]. Therefore, we argued that Lazy Test could be heuristically detected by comparing the effective operations of two test cases, stripping any test assertions. If the operations were identical, the test cases would be deemed to exercise the same production behavior and therefore be lazy, illustrated by Algorithm 4.1.

```
   Data: Set of test cases T = {t_1, t_2, ..., t_n}
   Result: Test cases marked with the Lazy Test smell
1  foreach (t_i, t_j) ∈ T × T, i < j do
2  |  A ← StripAssertions(EffectiveOperations(t_i));
3  |  B ← StripAssertions(EffectiveOperations(t_j));
4  |  if IdenticalOperations (A, B) then
5  |  |  Report t_i as lazy;
6  |  |  Report t_j as lazy;
7  |  end
8  end
```

Algorithm 4.1: Detection of Lazy Test Smell

Test Code Duplication

The Test Code Duplication smell was, similarly to Lazy Test, detected by comparing test case list representations with each other. Test cases that were very similar to at least one other test case were considered duplicated, and were detected as described by Algorithm 4.2. The similarity threshold of 80% was selected for detecting duplication, meaning that a test case was only considered duplicated if it was 80% or more similar with any other test. Multiple thresholds were experimented with, and 80% was found to provide the most meaningful results after inspecting the identified test cases.

```
   Data: Set of test cases T = {t_1, t_2, ..., t_n}
   Result: Test cases marked with the Test Code Duplication smell
1  foreach (t_i, t_j) ∈ T × T, i < j do
2  |  A ← DirectOperations(t_i);
3  |  B ← DirectOperations(t_j);
4  |  if Similarity (A, B) ⩾ 80% then
5  |  |  Report t_i as duplicated;
6  |  |  Report t_j as duplicated;
7  |  end
8  end
```

Algorithm 4.2: Detection of Test Code Duplication Smell

To compute similarity between lists of operations, we chose longest common subsequence as the base metric. The longest common subsequence of two sequences (lists of operations in this case) is defined as the longest subsequence that can by constructed by selecting elements existing in both sequences in order. For example, the longest common subsequence of the strings "ABCDE" and "CDEFG" is "CDE". A long common subsequence indicates that two sequences have much in common. Likewise, the difference between the sequence lengths and longest common subsequence length represents how dissimilar the sequences are. Factoring in both measures, we used the normalized similarity formula

$$\text{Similarity}(A, B) = \frac{2 \cdot |\text{LCS}(A, B)|}{|A| + |B|} \in [0, 1]\,, \tag{4.5}$$

where A and B are lists of operations. In the example of "ABCDE" and "CDEFG", their similarity would be computed as 60%.

When computing the similarity between test cases, list representations of direct operations were used over effective operations, meaning that code duplication was only considered for test case source code and did not consider duplicated helper function source code. An attempt to identify duplication in effective operations was done, but resulted in very high similarity levels overall as many helper functions were frequently used, and the size of the helper functions dominated the test cases.

4.3.7 Data analysis

Due to the small sample size of eight RAFTs and three RIFTs obtained from the investigation of RQ1, performing statistical testing on the distribution of test smells and code metrics among the flakiness labels (non-flaky, RAFT, RIFT) was infeasible. Instead, the extracted test smells and code metrics were analyzed qualitatively, categorized by test case labels. Each metric and smell was visualized in relation to the labels to identify differences in distribution between label groups (Appendix D). For instance, Figure D.1 illustrates the frequency of the Sleepy Test smell for each label, while Figure D.8 shows the distribution of Effective Lines of Code across the labels.

Summary: This chapter provided a detailed explanation of the methodology used in this work. The methods were presented in alignment with the research questions, starting with RQ1, which covered the dataset, test execution framework and statistical analysis. For RQ2, two approaches to correlate root causes with resource impact were presented. Lastly, for RQ3, the collection and analysis of test smells and code metrics was introduced. The next chapter presents the results of these methods without extensive analysis, as they will be discussed in the Discussion chapter.

---

5 Results

The results chapter presents all relevant findings of the research questions, in order. The results are interpreted without deeper analysis to provide a clear and objective view.

5.1 Computational resources and test flakiness

RQ1: How does computational resource availability correlate with test flakiness?

In this section we present insights gained from the experiment conducted to answer RQ1. Section 5.1.1 presents the identified flaky tests and their failure rates in relation to CPU usage. Section 5.1.2 describes the flaky tests in more detail, classifies them as RAFT or RIFT and presents the results of the statistical analysis to support the classifications. Finally, Section 5.1.3 provides a comparison of results between running tests sequentially and in parallel.

5.1.1 Experiment observations

The experiment was conducted by executing the selected test suites 23,000 times. The test case with the most executions therefore had 23,000 executions, while some test cases were executed less than 15,000 times. This was partly because some test suites were disregarded after thousands of executions since they did not contain a single test case that had failed for any of these runs. The other reason why the number of executions per test case differed is that when a test case failed, the execution was interrupted and no subsequent test cases were run. Figure A.1 shows how the executions of the test cases were distributed. We can see that 174 of the 199 test cases were executed more than 20,000 times while the remaining 25 test cases were run less than 15,000 times. These 25 test cases were part of test suites that were found to not be flaky, early in the experiments.

From these executions, 56 test cases had been observed to fail at least once. 11 of these test cases obtained a failure rate of at least 0.1% for at least one of the five CPU usage intervals introduced in Section 4.1.5. These 11 test cases are hereafter referred to as the flaky tests. They are shown in Figure 5.1 and analyzed further in Section 5.1.2. These test cases were all executed more than 20,000 times.

Figure 5.1 illustrates the failure rate that each of the flaky tests exhibited under different CPU usage levels in the form of a heatmap. Each column represents one of the five CPU

usage intervals and each of the eleven rows represent a flaky test. The values in the boxes range from 0.0, shown in white, to 6.7, shown in dark red. The same information is provided in the form of bar charts in Appendix B, where each flaky test is presented in their own figure.

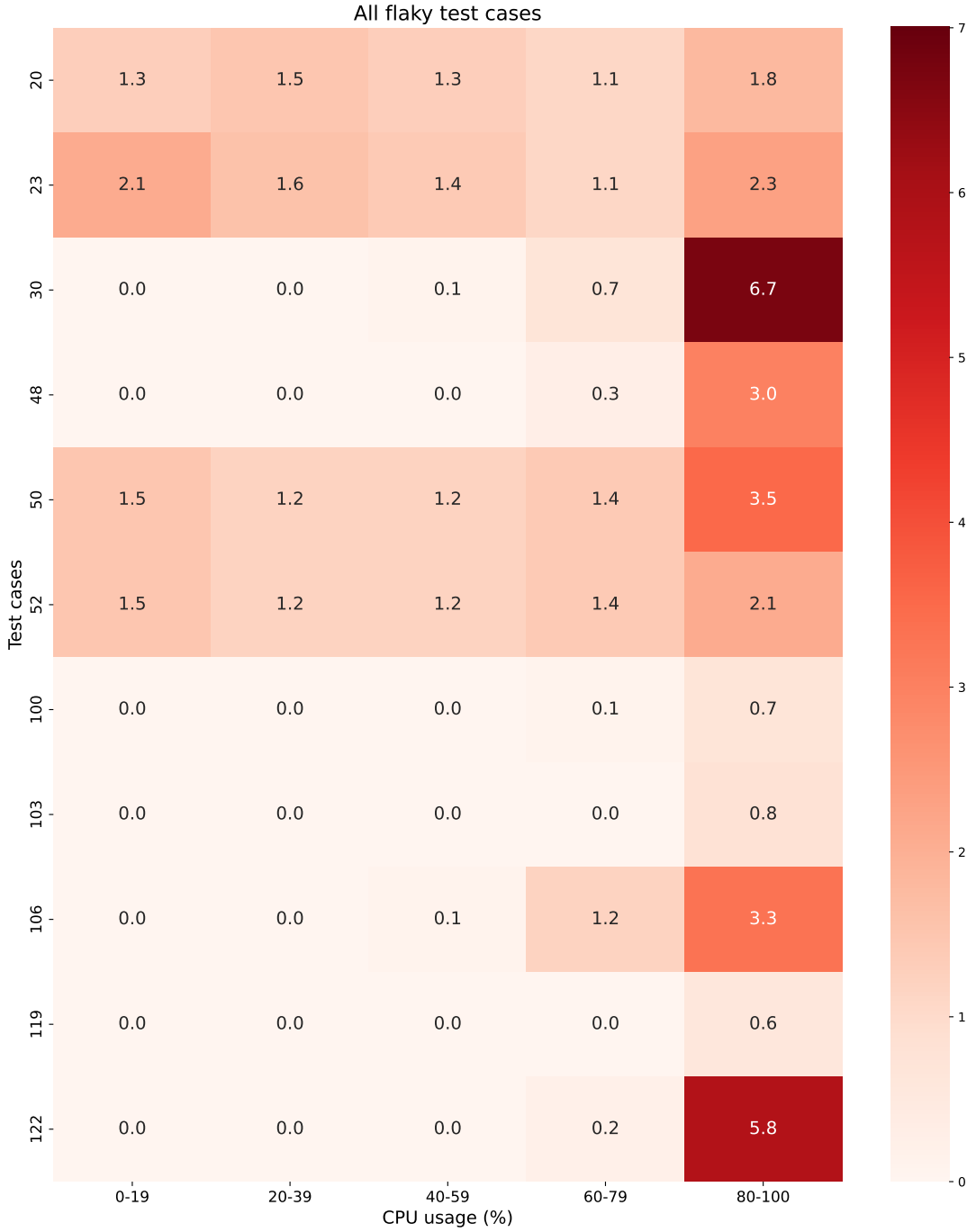


Figure 5.1: Failure rates of all flaky test cases (%)

The general trend in these results is that the highest failure rate can be found for the highest CPU usage, indicating that executions with CPU usage over 80% caused these test cases to fail more often than other executions. However, the failure rate was not increasing for every interval step for flaky tests 20, 23, 50 and 52. Seven of the flaky tests had a failure

rate of 0.1% or lower for CPU usage lower than 60%, while the remaining four flaky tests had a failure rate of 1.1% or higher for all ranges of CPU usage.

5.1.2 Test case flakiness classifications

This section presents the test cases based on similar patterns in the distribution of failure rates over CPU usage. Each flaky test was classified as either RIFT or RAFT. The full classification can be seen in Table 5.1.

Test cases 30, 48, 106 and 122 had failure rates of 0.1 or lower for CPU usage below 60%. For the CPU usage of 80% or more, the failure rate increased to 3.0% or more. The highest failure rate in this category was 6.7% for Test Case 30. This group of test cases were classified as RAFT since high resource usage caused the failure rate to increase drastically.

Test cases 100, 103 and 119 similarly had increased failure rate for high CPU usage compared to low CPU usage, however, these flaky tests did not reach a failure rate over 0.8%, even for high CPU levels. These flaky tests are also classified as RAFT since the trend was still the same, that the failure rate increased when the CPU usage increased.

Flaky test 50 was a special case in these results. The failure rate for all CPU usage intervals except the highest was stable around 1.3%, but went up to 3.5% failure rate for the highest CPU usage interval. This case differs from the flaky tests mentioned previously because it does not have a failure rate of 0.1% or lower for low CPU usage interval, however, it does seem to be affected by the CPU usage. This test case was also classified as a RAFT.

The remaining test cases 20, 23 and 52 were classified as RIFT since the failure rate for the different CPU usage intervals do not appear to increase significantly from the lowest CPU usage interval to the highest CPU usage interval. The failure rate for flaky test 23 even slightly decreased as the CPU usage increased. These three test cases do not exhibit a strictly non-decreasing trend compared to the other flaky tests.

Statistical analysis

By using the one-sided Wilcoxon rank-sum test on the test cases, we found that eight out of the eleven flaky tests could be determined to be RAFT. The remaining three tests were classified as RIFT. The null hypothesis that the distribution of CPU values for passed and failed executions were the same could not be rejected for these three test cases. The test cases that were shown to be RIFT were all part of the group mentioned earlier that had a failure rate of at least 1.1% regardless of CPU usage interval.

Table 5.1: Classification of flaky tests

Test case	Classification
20	RIFT
23	RIFT
30	RAFT
48	RAFT
50	RAFT
52	RIFT
100	RAFT
103	RAFT
106	RAFT
119	RAFT
122	RAFT

5.1.3 Impact of parallelism

To ensure that running the tests in parallel did not have an impact on the results, we ran the experiment with different levels of parallelism and visualized the flakiness for each (Appendix C). For instance, each subplot in Figure C.1 represents a different level of parallelism for Test Case 20: "unfiltered" indicates that all executions are included while "15 in parallel" only include executions which were run as 15 in parallel. Generally, running 30 tests in parallel resulted in high CPU usage even without additional simulation, making it impossible to execute tests at CPU usage levels between 0–20% for this level of parallelism. With the exception of test cases 20, 23, 52 and 100, every test case clearly had similar distributions of flaky failures over CPU usage for each level of parallelism.

The test cases 20, 23 and 52, shown in Figures C.1, C.2 and C.6, appeared to fail slightly more for low CPU usage levels and when run with no parallelism, in comparison to failing similarly for higher CPU usage levels when run with 15 or 30 tests in parallel. Since these tests exhibited failures for all levels of CPU usage and parallelism, we considered them as resource-independently flaky with regards to CPU usage, and therefore resource-independent in the scope of this thesis.

Test Case 100, shown in Figure C.7, only failed for higher CPU usage levels under higher parallelism, meaning that parallel executions allowed us to identify an additional resource-affected flaky test.

RQ1: Out of 199 investigated tests, we found 8 resource-affected flaky tests (RAFTs) and 3 resource-independent flaky tests (RIFTs).

5.2 Test flakiness root causes

RQ2: How do specific root causes of flaky tests and computational resource availability contribute to flakiness?

The results of the two methods described in Section 4.2 are presented here. Section 5.2.1 describes the subject matter expert's analysis of test logs generated during the RQ1 experiment. Section 5.2.2 presents the results of the survey and experiments on the fixes of flaky tests found in version control history.

5.2.1 Analysis of test logs

The subject matter expert categorized the root causes of the 11 test cases as follows: three were attributed to Async Wait, three to Concurrency issues, and the remaining five were grouped under Other. The "Other" category included cases where the root cause did not align with established categories or could not be determined. The following sections provide detailed explanations for each root cause category.

Async Wait

The test cases flagged as Async Wait were associated with applying various configurations to the system under test (SUT). These configurations were applied asynchronously, and the test driver relied on a fixed wait time before initiating execution and making assertions. However, if the allotted time was insufficient during a particular test run, the execution could start before the configuration was fully applied, leading to test failures. The subject matter expert concluded that an effective solution for this issue would be to implement a mechanism in the SUT to signal the test driver once the configuration process is complete.

Figure 5.2 presents a heatmap illustrating the failure rates of the three test cases labelled as Async Wait, sorted by CPU usage. For example, the value 6.7 in the top right corner indicates

that test case 30 exhibited a failure rate of 6.7% under conditions of 80–100% average CPU usage. The figure reveals a positive correlation between failure rates and CPU usage for these test cases. Specifically, test cases 30 and 122 showed failure rates exceeding 5% when CPU usage was 80% or higher, while failure rates for other conditions were below 1%. Notably, Test Case 119 had a maximum failure rate of 0.6%, significantly lower than the others, with failure rates below 0.1% at CPU usage levels under 80%.

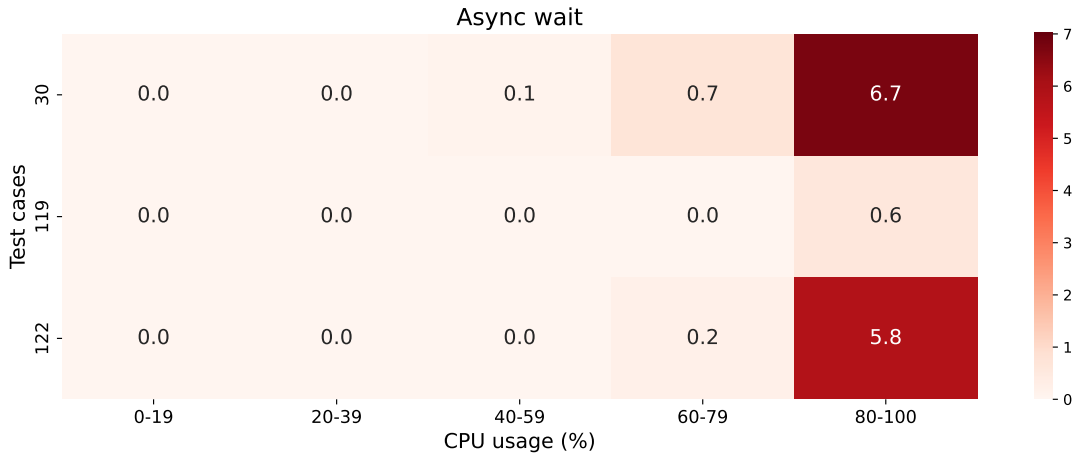


Figure 5.2: Failure rates of test cases with the flakiness root cause Async Wait (%)

These three flaky tests were categorized as RAFT in Section 5.1.2. The data confirms a positive correlation between failure rates and CPU usage for these tests.

Concurrency

The test cases that failed due to Concurrency issues made assumptions about the order in which events would occur during execution. Since the system was multithreaded, the order of events was non-deterministic, although some event sequences occurred more frequently than others. Regular event orders — those that occurred more often — typically resulted in passing tests, while irregular event orders led to failures.

The three test cases identified as having Concurrency-related issues were the only flaky tests categorized as RIFT in Section 5.1.2. The failure rate of these tests did not show a correlation with CPU usage. Statistical analysis found no significant difference in the CPU usage distributions between failed and passed executions, and as illustrated in Figure 5.3, no increasing trend in failure rates with higher CPU usage was observed for these test cases.

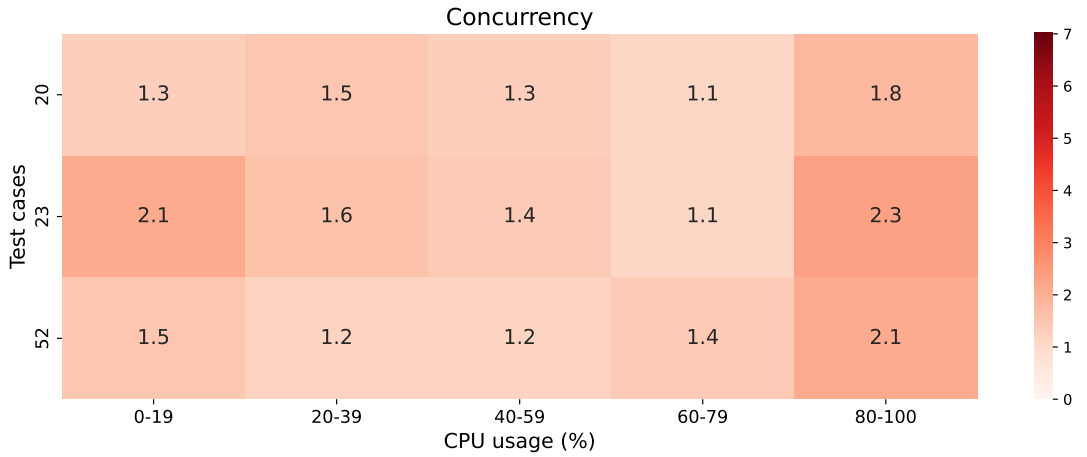


Figure 5.3: Failure rates of test cases with the flakiness root cause Concurrency (%)

Other

The remaining five test cases were categorized as "Other" because no specific root cause category was deemed appropriate. Test cases 48 and 50, shown in Figure 5.4, were related to Concurrency but did not fully fit that category. In these cases, the SUT exhibited non-deterministic behavior at the component level, yet still produced the expected result at the system level. This non-determinism caused the tests to fail due to assumptions made about the SUT. While similar to Concurrency-related issues, the failure was not caused by race conditions or synchronization problems. Instead, the test failed because it expected a deterministic outcome that did not account for the variability in how individual components behaved under certain conditions.

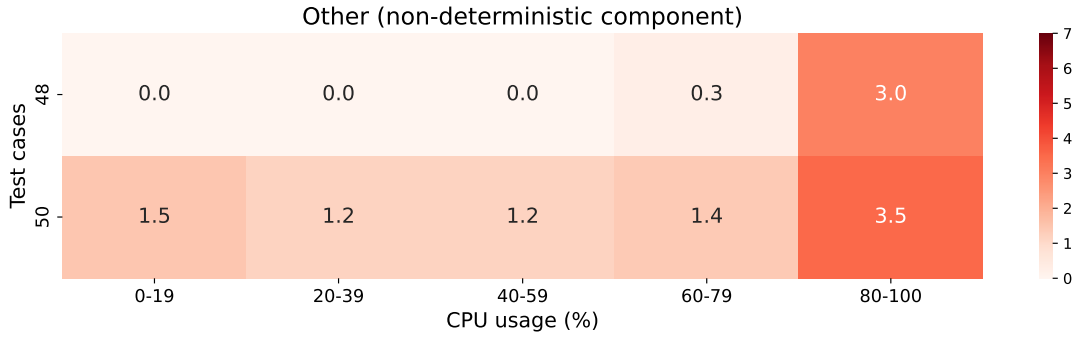


Figure 5.4: Failure rates of test cases with the flakiness root cause Other, described as non-deterministic component (%)

An example of this issue is shown in Listing 5.1. In this case, some data could be sent in different ways within the component, depending on specific conditions. The test case expected the data to be sent using one format, referred to as "`Data_A`". However, when the data was sent using a different format, "`Data_B`", the test failed. Although conditional logic could have been employed to account for both formats, this approach was not considered good practice and was not implemented in the testing framework. A better solution would have been to standardize the data format before making assertions, ensuring that the SUT behaved more deterministically. This approach would guarantee consistent outputs, assuming the other parts of the system are not flaky.

Test cases 100, 103, and 106, shown in Figure 5.5, were categorized as "Other" due to the following issue: the tests fail when an expectation is met before it is explicitly set, causing

```
1  // SUT can send two different messages
2  void triggerBehavior() {
3      if (condition) {
4          sendMessage("Data A");
5      } else {
6          // Data B is correct under these conditions and will lead to the same
               behavior for the system
7          sendMessage("Data B");
8      }
9  }
10
11 // Expects the most common message to be sent
12 TEST(TriggerDataSending) {
13     triggerBehavior();
14
15     expectDataA();
16     synchronize();
17 }
```

Listing 5.1: Simplified example of non-deterministic component

the system to experience an unexpected event. This behavior is flaky because the event could occur later, which would prevent the test from failing in those cases. To illustrate this, Listing 5.2 provides a simplified example. The test case triggers a behavior in the SUT that causes two events: FastData is collected and sent, and SlowData is collected and sent. These actions occur concurrently, with the fast data typically being sent before the slow data. The test waits for all expectations introduced before line 6 to be met using the synchronize statement on line 6. The test fails if the slow data is sent before the fast data, but in the usual case, the fast data is sent first, the test synchronizes, the slow data is sent, and the test synchronizes again. While the issue could be resolved by removing the synchronize statement, real-world test cases often involve more complex scenarios where the solution is not as straightforward. Similar to the Async Wait test cases, the flaky tests categorized as "Other" exhibited significantly different CPU distributions between passed and failed runs. All flaky tests with "Other" as the root cause were classified as RAFT.

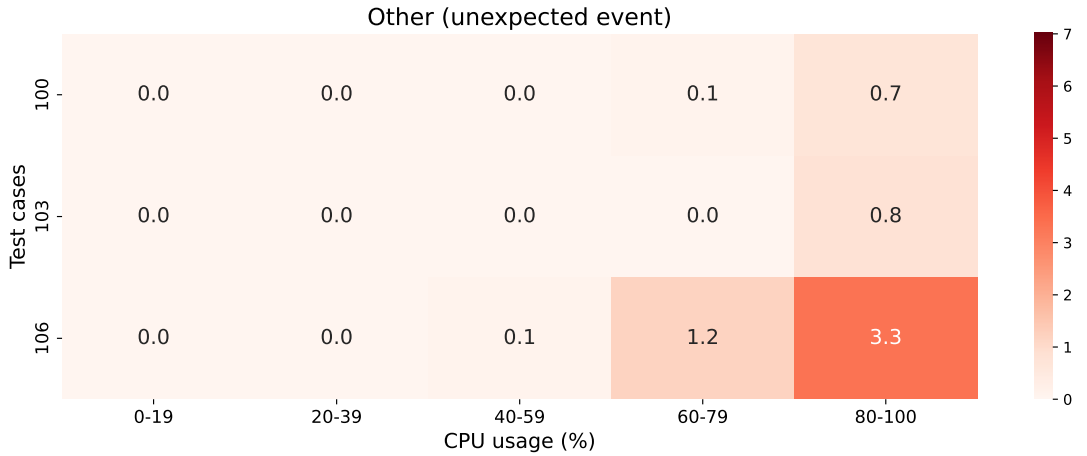


Figure 5.5: Failure rates of test cases with the flakiness root cause Other, described as unexpected event (%)

```
// Expects the two events to happen in order from one trigger
TEST(TriggerTwo) {
    triggerBehaviorFastAndSlow();

    expectFastData();
    synchronize();

    expectSlowData();
    synchronize();
}
```

Listing 5.2: Simplified example of unexpected event order

5.2.2 Fixes of flaky tests

The search in the version control history of the project yielded 63 commits. The authors of the commits were identified and surveys were sent out to each of them with each of their found commits. One of the 63 commits was not categorized since the author did not know how to categorize the root cause that was fixed in the commit. The other commits were categorized as shown in Table 5.2.

Table 5.2: Labelled Test Flakiness Root Causes for commits in Version Control

Root Cause	Number of Instances
Concurrency	19
Test Order Dependency	5
Async Wait	37
Unordered Collection	0
I/O	0
Time	0
Test Case Timeout	1
Test Suite Timeout	0
Other	2

6 commits were chosen from the categorized commits with 3 from the Concurrency category and 3 from the Async Wait category. At the start of the experiments, these commits were chosen from the same source code component, but was later changed due to many of the tests not being able to run in parallel, which lowered the number of possible executions for the experiments. Async Wait Commit #3 was not able to be run in parallel but was used since it fixed two test cases instead of one, compared to the other five commits. The final distribution of commits were the following: one Async Wait commit was related to the source code component that was the main focus in RQ1, and the rest of the commits were related to another component. These two components were similar enough that the experimentation implementation did not need changing.

Concurrency and Async Wait were the two most common categories and were therefore chosen for the experiments. This selection had several benefits: high frequency in this study could indicate a high frequency for the category in the company as a whole and therefore be meaningful for future fixes, and choosing categories with many instances lead to more options when it came to source code components and how many tests could run in parallel during an execution. The third most common root cause in this survey was test order dependency which was thought to not be interesting to investigate since we did not reorder test cases between test executions. Two categories were chosen instead of one to be able to compare the results between them and draw conclusions about their different behaviors.

Async Wait Commit #1 was chosen to verify if running one test case affected the outcome of the failure rate. The test suite that this commit was related to was executed 2400 times.

The results are shown in Figure 5.6. The failure rate was unchanged for high CPU usage compared to Figure 5.7 but increased by 0.08 percentage points for low CPU usage. This change was equivalent to one more failure, going from 0 to 1 for this CPU interval.

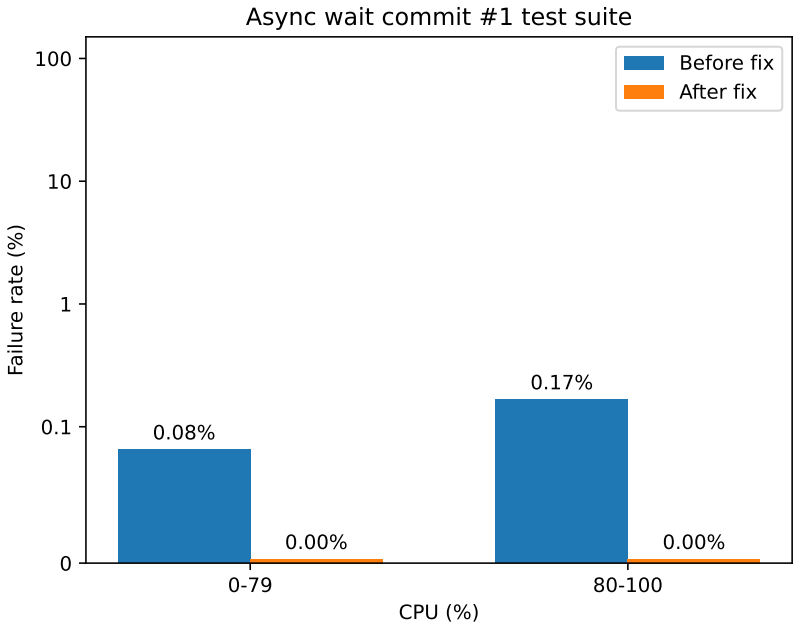


Figure 5.6: Async Wait Commit #1 test case failure rate for suite execution (%)

The final classification is presented in Table 5.3. The possible classifications were RAFT, RIFT and Non-flaky. Each test case was classified both before and after the commit. The two root causes appear to have similar results when it comes to their classifications before and after their respective commits, where every test case was resource affected before the commit and about half were not flaky after the commit. No instances of RIFT were found in this dataset. The two root causes and their respective commits are analyzed in further detail in the next sections.

Table 5.3: Classification of test cases in each commit before and after their respective changes

Test case	Before commit	After commit
Async Wait #1	RAFT	Non-flaky
Async Wait #2	RAFT	RAFT
Async Wait #3.1	RAFT	RAFT
Async Wait #3.2	RAFT	Non-flaky
Concurrency #1	RAFT	RAFT
Concurrency #2	RAFT	RAFT
Concurrency #3	RAFT	Non-flaky

Async Wait

In Figures 5.7 to 5.10 we observe the failure rate for the test case or test cases involved in the commit. For Async Wait Commit #3, two test cases were aimed to be fixed. For that commit, the failure rate for each of the two test cases are presented in separate plots. The plots show the failure rate before and after the supposed fix was introduced, as shown in blue (before) and orange (after). The results are also split based on the average CPU usage on the machine during the test execution, not the CPU load that the framework introduced.

The commits labelled to fix an Async Wait test flakiness root cause appeared to fail more often for CPU usage over 80%. Async Wait #1 only failed 2 times in total, and the failures were for high CPU usage before the implemented fix. The test case was therefore labelled as RAFT before the commit and as Non-flaky after the commit because it did not exhibit any flaky behavior after the fix. Async Wait #2 was deemed to be resource affected both before and after the commit. This test case was the only Async Wait test case that failed for low CPU levels. However, it only failed one time in 1200 executions and therefore did not qualify as flaky based on the definition used in this thesis. For high CPU usage, the fix decreased the failure rate with about 40% of the original value. Async Wait #3.1 had similar results as Async Wait #2. Although its failure rates before and after the commit was much lower in comparison, it was classified as RAFT since the CPU usage seemed to have an impact on the failure rate. Async Wait #3.2 did not fail for low CPU usage and only had a failure rate of 0.08% for high CPU after the fix. This meant that the test case was classified as Non-flaky after the commit and it was classified as RAFT before the commit.

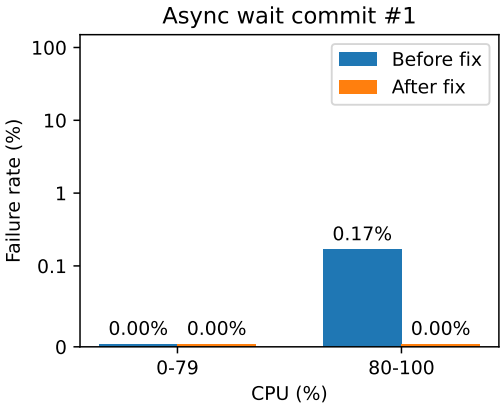


Figure 5.7: Async Wait Commit #1 test case failure rate (%)

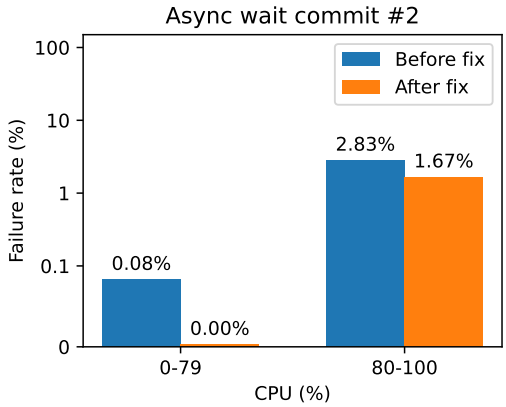


Figure 5.8: Async Wait Commit #2 test case failure rate (%)

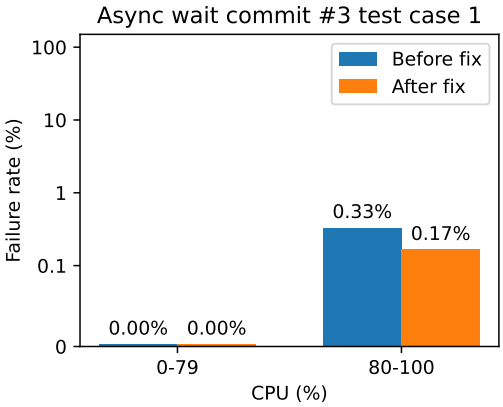


Figure 5.9: Async Wait Commit #3 Test Case #1 failure rate (%)

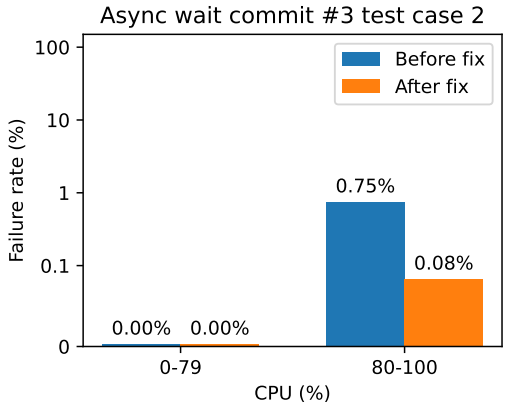


Figure 5.10: Async Wait Commit #3 Test Case #2 failure rate (%)

Concurrency

The commits labelled to fix Concurrency related issues can be seen in Figures 5.11 to 5.13. They had the same general trend as Async Wait, where the failure rate increased for high CPU usage, although the Concurrency commits exhibited more varied results among themselves.

Concurrency Commit #1 seemed to be resource affected since it did not fail for low CPU usage levels but had a failure rate over 0.1% for higher CPU usage. The implemented change did not appear to fix the flakiness completely since the test still failed after the fix was implemented. Concurrency Commit #2 appeared to fail more often for high CPU levels after the implemented fix. 40 out of the 41 failures were due to a different reason than what was described in the commit and related task description. All of the failures before the fix were due to the problem described in the commit. Concurrency Commit #3 stood out from the other commits due to the high failure rate. For high CPU usage levels, the test case failed almost half of the time. For low CPU usage, the test case failed 3 times during the 1200 executions. No failures were identified after the fix for any CPU usage.

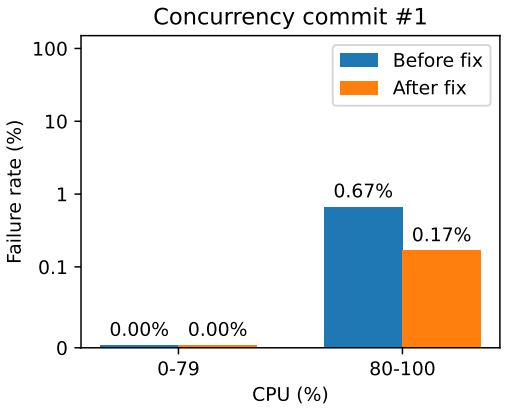


Figure 5.11: Concurrency Commit #1 test case failure rate (%)

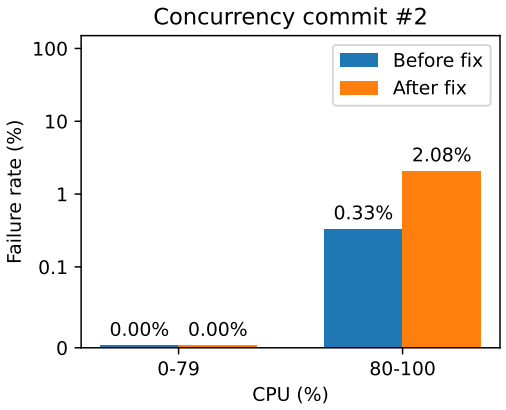


Figure 5.12: Concurrency Commit #2 test case failure rate (%)

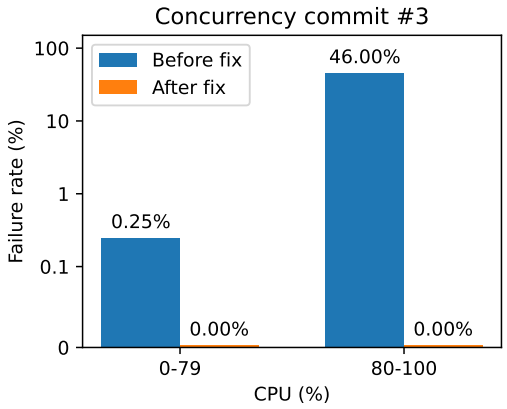


Figure 5.13: Concurrency Commit #3 test case failure rate (%)

RQ2: We found that flakiness caused by Async Wait was affected by CPU usage in all cases, while Concurrency was affected in only some cases.

5.3 Test smells and code metrics

RQ3: How do test smells, code metrics and computational resource availability predict test flakiness?

During the investigation of RQ3, test smells and code metrics were extracted from a dataset of 199 functional integration test cases. Test cases in the dataset had previously been labelled with 188 non-flaky tests, 8 resource-affected flaky tests (RAFT) and 3 resource-independent flaky tests (RIFT) from RQ1. First, we present the smell and metric characteristics overall, in Section 5.3.1. Then we analyze individual labels: non-flaky test cases (Section 5.3.2), RAFTs (Section 5.3.3) and RIFTs (Section 5.3.4). Finally, a comparative analysis is made between the labels in Section 5.3.5. Tables 5.4 and 5.5 summarize the findings.

Table 5.4: Results summary: test smells

Test Smell	Findings
Sleepy Conditional	Most test cases exhibited Sleepy Test (91%) and Conditional Test Logic (89%). Their 100% appearance in flaky tests suggests them as weak predictors for flakiness. (Figures D.1 and D.2).
Eager	Some test cases exhibited Eager Test (30%). Its absence from RIFTs suggests it as a possible predictor for non-RIFTs (Figure D.3).
Lazy	Few test cases exhibited Lazy Test (18%). Its high frequency in RIFTs (67%) suggests it as a predictor for RIFTs, while its absence from RAFTs suggests it as a predictor for non-RAFTs (Figure D.4).
Duplication	Most test cases exhibited Test Case Duplication (59%). Its low frequency in RAFTs (25%), suggests it as a predictor for non-RAFTs (Figure D.5).

Table 5.5: Results summary: code metrics

Code Metric	Findings
LOC CC	The distributions of Lines of Code, Effective Lines of Code and Effective Cyclomatic Complexity were shifted in RAFTs compared to non-flaky tests, allowing for the selection of a minimum threshold that disproportionally exclude more non-flaky tests than RAFTs. RIFT was shifted in the other direction, allowing a maximum threshold for capturing RIFTs. Direct Cyclomatic Complexity was generally low for all labels (Figures D.6 to D.13).
Sleeps Asserts Syncs	All direct and effective metrics, except for direct sleeps, followed the same distribution pattern as LOC and CC, shifting toward higher values for RAFTs and lower values for RIFTs. Number of direct sleeps were generally zero for all labels (Figures D.14 to D.25).

5.3.1 All test cases

Looking at all 199 test cases, including both flaky and non-flaky, the smells Sleepy Test, Conditional Test Logic and Test Code Duplication stood out. These smells appeared with high frequencies, and were present in a majority of test cases, shown in Table 5.6. Specifically, Sleepy Test was the most frequent, present in 91% of all test cases, indicating that thread-sleep logic was a common characteristic of the investigated tests, pointing to reliance on constant-time thread sleeps as a means to asynchronously wait for SUT behaviors to complete before progressing a test. Similarly, Conditional Test Logic appeared in 89% of cases, suggesting most tests to include conditional branching logic. The presence of Test Code Duplication in 59% of cases indicated that a majority of cases shared at least 80% of direct operations (in order) with another test case.

Table 5.6: Test smell frequencies for all test cases

Test Smell	Frequency	Relative Frequency
Sleepy Test	171	0.91
Conditional Test Logic	168	0.89
Eager Test	56	0.30
Lazy Test	33	0.18
Test Code Duplication	110	0.59

In contrast, Eager Test and Lazy Test appeared less frequently, with relative frequencies of 30% and 18% respectively. While less frequent, the presence of these smells indicated instances of tests exercising multiple SUT behaviors per test case (Eager Test) and tests exercising the same or very similar SUT behavior over multiple cases (Lazy Test).

The metrics in Table 5.7 provide further insight into the structural characteristics of the test cases. On average, test cases consisted of 46 lines of code, typically ranging from 21 to 61 lines (25th-75th percentile). The average effective Lines of Code was 197, more than four times the number of direct Lines of Code, indicating that most of the relevant content of a test was written in its used helper functions. Cyclomatic Complexity, with an average of 2, was relatively low for most test cases. However, a minority of cases (between 5% and 25%) had a high Cyclomatic Complexity of 10 or more, also shown in Figure D.10, indicating a subset of moderately complex test cases. The large difference between direct and effective Cyclomatic Complexity, average 2 vs. 14, highlights the fact that the detailed logic of many test cases was bound to helper functions rather than direct test code, and matches the similar relation between direct and effective Lines of Code.

Table 5.7: Code metrics mean, standard deviation and percentiles for all test cases

Metric	Mean	STD	5%	25%	50%	75%	95%
Lines of Code	46	31	11	21	42	61	105
Effective Lines of Code	197	102	14	134	204	258	362
Cyclomatic Complexity	2	3	1	1	1	2	10
Effective Cyclomatic Complexity	14	8	1	8	14	17	28
Number of Sleeps	0	1	0	0	0	0	3
Effective Number of Sleeps	2	2	0	1	2	3	5
Number of Asserts	2	3	0	0	1	2	8
Effective Number of Asserts	16	10	1	9	15	22	35
Number of Synchronizations	1	1	0	0	1	2	4
Effective Number of Synchronizations	4	3	0	2	3	5	9

Most test cases (>75%, Table 5.7) did not directly contain thread-sleeps, while effective thread-sleeps were present 91% of test cases, as shown by the Sleepy Test smell in Table 5.6. Test cases similarly had a general low number of direct assertions and synchronizations (average 2 and 1), with the effective metric being dominant. The number of asserts and synchronizations naturally follow the same patterns due to the nature of the test environment (Section 4.1.2) where each set of assertions was followed by a synchronization.

5.3.2 Non-flaky test cases

The non-flaky test cases showed nearly identical smell frequencies (Table 5.8) and metrics (Table 5.9) as the overall test cases. This similarity is explained by the imbalanced dataset, where non-flaky tests made up the vast majority (188 out of 199 cases). Because of this imbalance, the analysis of the overall test cases is practically an analysis of the non-flaky subset.

Table 5.8: Test smell frequencies for non-flaky test cases

Test Smell	Frequency	Relative Frequency
Sleepy Test	160	0.90
Conditional Test Logic	157	0.89
Eager Test	53	0.30
Lazy Test	31	0.18
Test Code Duplication	106	0.60

Table 5.9: Code metrics mean, standard deviation and percentiles for non-flaky tests cases

Metric	Mean	STD	5%	25%	50%	75%	95%
Lines of Code	46	32	11	20	41	62	108
Effective Lines of Code	197	105	14	134	204	258	362
Cyclomatic Complexity	2	3	1	1	1	2	11
Effective Cyclomatic Complexity	14	8	1	7	14	17	28
Number of Sleeps	0	1	0	0	0	0	3
Effective Number of Sleeps	2	2	0	1	2	3	5
Number of Asserts	2	3	0	0	1	2	8
Effective Number of Asserts	16	10	1	10	15	22	35
Number of Synchronizations	1	2	0	0	1	2	4
Effective Number of Synchronizations	4	3	0	2	3	5	9

The fact that the overall test smells and code metrics were heavily biased towards those of non-flaky tests confirms that the observations were not disproportionately influenced by the small set of flaky test cases. However, it also highlights the need for larger or more balanced datasets, with more flaky test cases to differentiate smells and metrics between flaky and non-flaky cases.

5.3.3 Resource-affected flaky test cases

Analysis of the eight Resource-Affected Flaky Test cases (RAFTs) showed that all of them exhibited the Conditional Test Logic and Sleepy Test smells, as shown in Table 5.10. This suggests that the smells could be used as predictors for identifying RAFTs, but also that the presence of conditional logic and thread-sleeps could be a pre-condition for test flakiness.

Table 5.10: Test smell frequencies for RAFTs

Test Smell	Frequency	Relative Frequency
Sleepy Test	8	1.00
Conditional Test Logic	8	1.00
Eager Test	3	0.38
Lazy Test	0	0.00
Test Code Duplication	2	0.25

Conversely, few RAFTs exhibited the Lazy Test and Test Code Duplication smells. This indicates that the RAFTs in the dataset were generally different from other tests, including other RAFTs, with regards to code duplication and mostly exercised different SUT behaviors. The low frequencies of these smells also suggests their potential as predictors for identifying non-RAFTs when they are present.

The code metrics for RAFTs (Table 5.11) show mean values similar to that of overall test cases. However, the lower percentiles of the metrics (Effective) LOC, Effective CC, Effec-

tive Number of Asserts and Effective Number of Synchronizations are drastically higher for RAFTs than that of overall tests. For instance, the 5th-percentile of RAFT LOC is 42, which is the same as the 50th-percentile of overall LOC. This implies that removing all test cases with fewer than 42 lines of code would disproportionally eliminate more non-RAFTs than RAFTs. This finding suggests that LOC could potentially be used as a minimum threshold when predicting RAFTs. Figures D.6 to D.9 provide more detailed illustrations of this. The higher low-percentile values also suggest that RAFTs tend to consist of non-trivial logic, possibly suggesting maintainability metrics as suitable predictors for RAFTs.

Table 5.11: Code metrics mean, standard deviation and percentiles for RAFTs

Metric	Mean	STD	5%	25%	50%	75%	95%
Lines of Code	55	16	42	46	52	56	80
Effective Lines of Code	217	47	158	180	214	265	271
Cyclomatic Complexity	2	2	1	1	1	1	6
Effective Cyclomatic Complexity	14	4	10	12	14	15	20
Number of Sleeps	0	0	0	0	0	0	1
Effective Number of Sleeps	2	1	1	2	2	3	4
Number of Asserts	2	1	0	1	2	2	3
Effective Number of Asserts	19	7	11	16	18	22	30
Number of Synchronizations	2	1	1	1	1	2	3
Effective Number of Synchronizations	5	2	3	4	5	6	7

The effective sleep metric shows that RAFTs in the dataset consistently relied on thread-sleeps, which could explain the resource-affected flakiness. However, the effective sleep metric compared very similar to that of overall tests.

5.3.4 Resource-independent flaky test cases

Since the dataset contained only three RIFTs, their smells and metrics are presented here as absolute values rather than relative frequencies and percentiles.

In Table 5.12 and Table 5.13, test cases 20 and 23 look identical. Upon inspection of their source code, it was discovered that these test cases only differed in assertions and operands, such that their structure was identical. Because of this, both test cases were assigned the same smells and metrics, and were specifically assigned the Lazy Test and Test Code Duplication smells due to their similarity.

Table 5.12: Test smell occurrences per resource-independent flaky test case

Test Smell	Test Case 20	Test Case 23	Test Case 52
Sleepy Test	X	X	X
Conditional Test Logic	X	X	X
Eager Test			
Lazy Test	X	X	
Test Code Duplication	X	X	

Table 5.13: Code metrics per resource-independent flaky test case

Test Smell	Test Case 20	Test Case 23	Test Case 52
Lines of Code	14	14	21
Effective Lines of Code	110	110	129
Cyclomatic Complexity	1	1	1
Effective Cyclomatic Complexity	8	8	10
Number of Sleeps	0	0	1
Effective Number of Sleeps	1	1	3
Number of Asserts	0	0	0
Effective Number of Asserts	7	7	9
Number of Synchronizations	0	0	1
Effective Number of Synchronizations	1	1	3

All RIFTs exhibited both the Conditional Test Logic smell and the Sleepy Test smell, in line with their high overall frequencies. No RIFT showed the Eager Test smell.

The RIFTs stood out as relatively simple test cases, with all metrics falling near or below the 25th-percentile of the overall tests. This could specifically indicate LOC and CC being usable predictors of non-RIFTs, as they measure size and complexity of a test.

5.3.5 Comparative analysis

A comparative analysis of test smell frequencies between non-flaky tests, RAFTs and RIFTs, as summarized in Table 5.14 and detailed in Figures D.1 to D.5, showed variance in the frequencies of test smells across the labels. Every smell exhibiting at least some variance suggests associations between test smells and flakiness characteristics.

Table 5.14: Test smell relative frequencies per flakiness label

Label	Sleepy	Conditional	Eager	Lazy	Duplicated
Non-flaky	0.90	0.89	0.30	0.18	0.60
RAFT	1.00	1.00	0.38	0.00	0.25
RIFT	1.00	1.00	0.00	0.67	0.67

The Sleepy Test smell (Figure D.1) was observed in every flaky test, but also in 90% non-flaky tests. This could suggest a positive relation between the presence of constant-time thread-sleeps and test flakiness, although the high frequency in non-flaky tests limits the predictive power of the smell. The high overall frequency of Sleepy Tests is likely a symptom of the asynchronous test environment (Section 4.1.2).

Conditional Test smell (Figure D.2) appeared in all RAFTs and RIFTs, with a slightly lower frequency in non-flaky tests (89%). While its similar frequency in all labels may limit its information gain, it could possibly still be used to distinguish flaky tests from non-flaky ones.

Eager Test smell (Figure D.3) was slightly more frequent in RAFTs (38%) compared to non-flaky tests (30%). Notably, this smell was absent in all RIFTs, suggesting that it could be used as a differentiator between RAFTs and RIFTs.

In contrast, Lazy Test smell (Figure D.4) showed the highest frequency in RIFTs (67%), while absent in RAFTs and low-frequent in non-flaky tests (18%). This difference in frequencies suggests Lazy Test as a strong predictor for RIFTs.

Test Code Duplication smell (Figure D.5) was predominantly present in non-flaky tests (60%) and RIFTs (67%), while underrepresented in RAFTs (25%). This pattern suggests that Test Code Duplication could be used as a predictor for non-RAFTs.

The code metrics showed some potential as predictors of non-flaky tests, RAFTs and RIFTs. Looking at the mean metrics across the labels in Table 5.15, only a trend of lower

RIFT values could be established. However, the details illustrated in Figures D.6 to D.25 reveal a difference in the distributions of metrics across the labels.

Table 5.15: Code metric averages per flakiness label (effective metrics in parenthesis)

Label	LOC	CC	Sleeps	Asserts	Synchronizations
Non-flaky	46 (197)	2 (14)	0 (2)	2 (16)	1 (4)
RAFT	55 (217)	2 (14)	0 (2)	2 (19)	2 (5)
RIFT	16 (116)	1 (9)	0 (2)	0 (8)	0 (2)

The Lines of Code metric was drastically lower on average for RIFTs compared to both RAFTs and non-flaky tests, both when considering the effective and direct metric. This, in combination with the same pattern for Cyclomatic Complexity, suggests RIFTs to be smaller and logically simpler than other tests. Furthermore, inspecting the distributions of Lines of Code, Effective Lines of Code and Effective Cyclomatic Complexity (Figures D.7, D.9 and D.13) reveals a shift between RAFTs and non-flaky tests, where the lowest RAFT values are drastically higher than the lowest non-flaky values. This shift suggests that each of the metrics has predictive power and could be used as a threshold when detecting RAFTs. For instance, selecting all tests with Effective Cyclomatic Complexity of at least 10 would eliminate all RIFTs and disproportionally more non-flaky tests than RAFTs (Figure D.13).

Each of the number of sleeps, asserts and synchronizations metrics (excluding direct sleeps) followed the same shifting pattern as for LOC, with higher minimum values for RAFTs. This suggests the metrics as usable predictors and associated with LOC and CC (Figures D.14 to D.25).

RQ3: We found a weak positive relation between flakiness and the smells Sleepy Test and Conditional Test Logic. Eager Test, Lazy Test and Test Code Duplication exhibited high variance across the labels non-flaky test, resource-affected flaky test (RAFT) and resource-independent test (RIFT). Among code metrics, Lines of Code and Cyclomatic Complexity occurred with a high minimum threshold in RAFTs relative to other tests, while being low for all RIFTs.

Summary: This chapter presented the results of the applied methods, highlighting key decision points and outcomes for each research question. For RQ1, the dataset contained 8 RAFTs and 3 RIFTs. In RQ2, Async Wait was found to be resource-affected, whereas Concurrency was only occasionally impacted. The results for RQ3 indicated that Eager Test, Lazy Test, Test Code Duplication, Lines of Code and Cyclomatic Complexity exhibited noticeably different results across different labels. The next chapter, Discussion, provides a deeper analysis of these results, evaluates the methods used and describes the work in a wider context.

6 Discussion

In this chapter, we discuss the results and methods, and position the work in a wider context. In Section 6.1, we analyze the key insights gained from the obtained results. In Section 6.2, we review the methods employed in this study and evaluate the sources used. The chapter concludes with Section 6.3, where we place the work within a broader context.

6.1 Results

In this section, we investigate the important findings, argue about how these results came to be and present the practical implications gained from the results of the study.

CPU impact on flakiness

RIFTs and RAFTs were found to behave quite differently for the same CPU usage intervals. Most of the resource-affected flaky tests (RAFTs) had a negligible failure rate for CPU usage below 60% and exhibited a sharp increase in failure rate for higher CPU loads. This conforms with the work of Silva et al. [5] who found instances of RAFTs where the failure rate increased by more than 200 times when put under resource constraints. Conversely, the resource-independent flaky tests (RIFTs) found in the dataset exhibited a somewhat stable failure rate over the CPU usage intervals. An interesting finding was that the failure rate sometimes decreased going from a lower CPU usage interval to a higher interval. A contributor to this could be that there was a lower number of runs with very low CPU usage compared to high CPU usage. The lowest number of runs for any interval among the RIFTs was 1687 for Test Case 52 in the lowest interval. This should still have been sufficient to get an accurate failure rate since the total number of failures in this interval was 27.

One finding was that most RAFTs have near zero failure rate for low CPU usage. Test Case 50 was the only exception. It was classified as RAFT, but it had a non-zero failure rate for low CPU usage intervals. The failure rate tripled for high CPU usage, which was a significant difference compared to the RIFTs found. This pattern could indicate some combination of root causes for the failures. If the test cases had two root causes, one that was resource-affected and one that was independent, they could amplify the failure rate. On the other hand, this argument could be made for any of the flaky tests, that the failure rate could be a combination of attributing factors.

Very infrequently failing tests

One interesting finding was that 45 test cases failed at least once but not more than 0.1% of the time for any of the five CPU ranges. These test cases failed between 1 and 4 times during the course of the experiment. It is possible but unlikely that these failures occurred due to environmental faults. This is because we were informed by subject matter experts that the testing framework did not use external resources and because environmental faults should affect parallel and subsequent executions. Another improbable reason would be that executions were measured incorrectly in the experiment. During the course of the experiment and while compiling the results of the experiment, the authors visually inspected log files to verify results and investigate experiment artifacts. A more probable explanation of these failed tests is that the testing framework or the system under test (SUT) had some kind of intrinsic non-determinism that manifests at non-deterministic moments of the execution. It could also be that the test cases were just flaky with a low failure rate. Lam et al. [47] found many flaky tests with failure rates lower than 0.1%, where the overall median failure rate was lower than 0.5%, which points to that flakiness can result in very low failure rates.

Identified flaky tests

All resource-independent flaky tests (RIFTs) exhibited failure rates below 3%. A reasonable assumption would be that RIFTs with higher failure rates would be quickly noticed and fixed by developers due to the negative impact of non-deterministic tests. RIFTs with low failure rates are less likely to trigger immediate action, allowing them to persist in the code base for a longer period of time. These characteristics could explain why only RIFTs with low failure rates exist in the dataset.

For resource-affected flaky tests (RAFTs), their low failure rates at low CPU usage are also not surprising. Seven RAFTs in the dataset exhibited failure rates below 0.1% when the CPU usage was under 40%. Depending on the hardware and workload of the environment where these tests are executed, the CPU usage would rarely exceed this threshold, leading to the test being perceived as stable. Conversely, the flaky tests that had been fixed also had a low failure rate. This indicates that they were found to be flaky despite the rarity of failures or that developers fixed them proactively, knowing that the implementation could be flaky.

The ease of fixing flaky tests likely influenced the composition of the dataset. If a test has a straightforward root cause, or is similar to previously resolved issues, then they could be addressed quickly after being identified. In contrast, tests that are harder to debug or reproduce can remain unresolved for longer. When these tests also have low failure rates or are triggered under specific resource conditions, they can also be perceived as lower priority. This aligns with Eck et al. [8], who surveyed developers and found that flaky test are often considered lower priority than permanently failing tests.

These dynamics collectively suggest that the dataset primarily represents flaky tests with subtle, resource-related causes or low failure rates that evade early detection and resolution.

CPU influence on concurrency

One major finding was that the only instances of resource-independent flaky tests (RIFTs) were attributed to the Concurrency root cause. This was found when the flaky tests from the experiment of RQ1 were analyzed, where the Concurrency root cause was present in all three RIFTs but was not the root cause of any RAFT. This was not the case when running experiments on commits that aimed to fix Concurrency issues, where the test cases instead exhibited resource-affected behavior. The datasets for each of the methods could be a reason for this difference. The fixes of flakiness are instances of tests that have been fixed whereas the flaky tests from RQ1 were tests that had not been fixed at the time when the experiment started. The flaky tests that remained in the code base had low failure rates for any CPU usage level which meant that they were more difficult to detect and could be considered as less of a

priority compared to test cases with high failure rate or simple solutions. Using Concurrency commit #3 as an example, it had a failure rate of 46% for CPU usage over 80%. It would be safe to assume that this test case was a priority to fix and would therefore not exist for long in the code base in its flaky state.

High frequency root causes

The most common root causes in the industry codebase were found to be Async Wait and Concurrency. Since the projects incorporated concurrent programming to a large extent, the prevalence of the Concurrency root cause was not surprising. Concurrency is generally seen as difficult, and mature projects using concurrent programming could create a high cognitive load for developers. The dominance of test flakiness attributed to Async Wait can also be explained as a consequence of the architecture of the projects. During the integration tests, the test driver communicated with the SUT asynchronously. This implementation could lead to new flaky tests arising before a synchronization mechanism is implemented. The general idea here is that waiting for a certain number of seconds is an easy solution to implement but will fail whenever the asynchronous task completes slower than expected.

Other root causes

Two different unspecified root causes were found and are described in the results of Section 5.2.1. The first mentioned was due to non-deterministic behavior on the component level and the second was due to an unexpected order of events. These root causes could fall into a category called “too restrictive range”, formulated by Parry et al. [4] as “Test where some of the valid output range falls outside of what is accepted in its assertions. This test is flaky, since it does not account for corner cases and thus it may intermittently fail when they arise”. This root cause matches the failures somewhat well. The non-deterministic component sometimes produced results that were valid but not expected in the test case, i.e. corner cases. The unexpected order of events also produced valid outcomes that fell outside of what was accepted in the test case assertions. “Too restrictive range” was not included in the list that was sent to the subject matter expert as it was not included in the original list of root causes defined by Luo et al. [1]. In the cases of non-deterministic components, the restrictive range is clear: the test driver expects one type of message but receives another, and the message is a valid output of the SUT. For the case of the unexpected message, it is not as obvious. However, Eck et al. [8] includes “improperly placed assertions statements” leading to failures, in their definition of the root cause. This fits in well with the description of the root cause given by the subject matter expert. Another root cause that could be applicable to non-deterministic components is the similarly named “non-deterministic specification”. This root cause manifests when a test makes assumptions about code that is specified to be non-deterministic. A common example is assuming a certain order when iterating over an unordered collection. This description fits non-deterministic components well since they can produce several valid outcomes and the test only expects one.

Higher frequency test smells

Among the test smells, Sleepy Test and Conditional Test Logic appeared in 89% and 91% of all tests respectively, but both also appeared with 100% frequency in flaky tests. These smells being more common in flaky tests is consistent with Camara et al. [11], which found Sleepy Test to be a strong flakiness predictor and Conditional Test Logic a weak one, though the overall high frequency is discrepant. One explanation for this difference could be that other researchers, to our best understanding, appear to only look at the immediate source code of test case functions [11, 31, 57]. We, on the other hand, based the smells Sleepy Test, Conditional Test Logic and Lazy Test on the effective source code of both test case functions

and related test helper functions (Section 4.3.6). The heavy use of test helper functions in our dataset — likely due to the complexity of the SUT — caused most interesting operations and patterns to be located in these helpers, rather than directly in the test cases. For instance, Figures D.14 to D.17 show that Effective Sleeps dominated direct Sleeps, and that redefining Sleepy Test using direct sleeps would have made only two flaky test cases exhibit the smell. Since the difference in number of sleeps between non-flaky tests, RAFTs (resource-affected flaky tests), and RIFTs (resource-independent flaky tests) was low (Figure D.15), direct sleeps provided little insight, and by including both direct and effective code we produced patterns that other tools could not have. It should be noted that use of test helper functions is considered a good practice for eliminating duplication [13, 30], and should therefore be expected. Furthermore, the high overall frequency of Sleepy Test can be explained by the asynchronous test environment utilized by each test case (Section 4.1.2). Test cases often had to perform actions in order, but could not always synchronously wait for one action to complete before triggering the next, making constant-time thread-sleeps necessary (e.g., Listing 4.3). Therefore, the high frequency of Sleepy Test was unsurprising, and the difference in frequencies between flaky and non-flaky tests suggests both Sleepy Test and Conditional Test Logic as (weak) predictors for identifying flaky tests.

Lower frequency test smells

Eager Test and Lazy Test appeared only in 30% and 18% of all tests, i.e., in a minority of tests. Eager Test, defined by a test exercising multiple SUT behaviors, appeared slightly more in RAFTs (38%) than non-flaky tests (30%) and never in RIFTs (Figure D.3). Lazy Test on the other hand, defined by one SUT behavior being exercised by multiple tests, appeared most in RIFTs (67%), in a few non-flaky tests (18%) and no RAFTs (Figure D.4). Both test smells were selected as they impact maintainability [13, 31, 30], and were therefore expected to cause flakiness through developer mistakes. The relatively low and similar frequency of Eager Test in non-flaky tests and RAFTs instead indicate it a poor predictor for either label. Based on its low frequency in RIFTs, Eager Test would best be used identifying non-RIFTs.

The complete absence of the Lazy Test smell in RAFTs could indicate it a useful predictor for identifying non-RAFTs. The disassociation between RAFTs and Lazy Test could be explained by the observed trend of RAFTs being more complex and therefore more dissimilar to other tests. Our detection rule for Lazy Test was to select any two tests with the same effective operations except for assertions (Algorithm 4.1), logically making the larger and more complex RAFTs less likely to be identical. This is further implied by the Test Code Duplication smell being underrepresented in RAFTs (Figure D.5), indicating RAFTs to be generally more unique than other tests. One could argue that duplicated flaky tests should be detected faster, and therefore easier to fix, because multiple instances of the same flaky code should fail more consistently. Since our detection method of Lazy Test and Test Code Duplication was based on code similarities and duplication, this could explain our findings of both smells being underrepresented in RAFTs. It could further be supported by industrial developers actively working to prevent flakiness, for example through this thesis, and therefore also fixing flaky tests as soon as they are detected, eliminating all easily detected flakiness.

The difference between RIFTs and RAFTs, with RIFTs exhibiting Lazy Test and Test Code Duplication, and RAFTs exhibiting Eager Test, is likely also explained by the difference in size and complexity between RIFTs and RAFTs. All observed RIFTs were much smaller compared to other tests (Figure D.7), while RAFT size instead was positively shifted. As mentioned, Lazy Test and Test Code Duplication are logically less likely in larger tests. Conversely, Eager Test was detected through the number of phases in a test case (Equation 4.4), and should therefore be more likely in larger test cases. These hypotheses provide reasonable explanations to our findings, but also open up the question of whether the insights gained from Lines of Code and Cyclomatic Complexity metrics make Lazy Test and Eager Test redundant when it comes to predicting resource-affected flakiness.

Maintainability code metrics

Inspecting the extracted maintainability metrics, i.e., Lines of Code and Cyclomatic Complexity, revealed shifted distributions between non-flaky tests, RAFTs and RIFTs. As already mentioned, RIFTs were found to be simple while RAFTs had a minimum threshold size, shown by Figure D.6. Direct Cyclomatic Complexity was an exception, as it was mostly zero across all labels, meaning that the complexity of practically every test case consisted of simple logic on the surface while their real complexity existed in helper functions. This suggests one must consider not only the test case source code but also that of its helper functions in order to capture the main logic. However, discussion with subject matter experts revealed that part of the helper functions did not affect flakiness, leading to us excluding certain functions from analysis. Therefore, project-specific insights are equally important when analyzing test flakiness. The shifted distributions of maintainability metrics between flakiness labels (e.g., LOC, Figures D.6 and D.7), indicate their usefulness for predicting flakiness, in line with the findings of Pontillo et al. [12], but also for distinguishing between resource-affected and resource-independent flakiness. The similar patterns between LOC and CC are logical as a higher LOC should imply higher CC. It is difficult to determine whether resource-affected flakiness is caused by conditional logic or simply becomes more likely with an increase in Lines of Code, when the two metrics were observed to be highly dependent. A dataset with higher variance between the two metrics could possibly be used to investigate this dilemma.

Operation-based code metrics

The operation-based code metrics, i.e., number of Sleeps, Asserts and Synchronizations revealed similar insights as the maintainability metrics, i.e., low values for RIFTs and higher for RAFTs. (Figures D.14 to D.25). Direct Sleeps, as mentioned, was the exception, generally being zero across all test cases. The strong relation with maintainability metrics imply that operation-based metrics have limited predictive power. This was both surprising but also logical: subject matter experts identified the combined use of asynchronous assertions, thread-sleeps and synchronization directives as a major cause of flakiness, but only a subset of patterns could become hazardous. Due to the complexity of these patterns we neither managed to manually or automatically detect them. We hypothesize that a more sophisticated approach, such as using language models to interpret our abstract code representations, could effectively recognize the hazardous patterns.

Other findings

Four of the seven commits that aimed to fix flakiness issues did not eliminate the occurrence of failures. For one of the commits, the failure rate even increased for high CPU usage after the fix was introduced. These fixes might have been made deliberately to only improve the failure rate but not fix the problems completely. In cases where the testing framework or the SUT is not designed in a way that would eliminate flakiness, changing this can take a long time and needs to be planned thoroughly. It would be reasonable to improve the failure rate in the mean time by making an easy change such as increasing the time for a thread sleep in the case of an Async Wait root cause for example.

One test case was found to fail more often after the attempted fix of flakiness was implemented. This test case was investigated by the authors. The conclusion was that at least all but one of the failures after the commit were not what the commit aimed to fix. This anomaly was most likely the product of general flakiness in the testing process and did not have to do with the specific commit or test case. The fact that the test case did not fail as often for the same reason before the changes were applied could be attributed to randomness of the distribution of CPU usage in the high CPU interval.

Practical implications

The results gathered appears to fit the narrative of the related research, that a substantial amount of flaky tests are affected by resource limitations [5]. Compared to Silva et al. [5], our dataset had a higher ratio of RAFTs (8/11 compared to 283/608) but a smaller number of flaky tests (11 compared to 608). This prevalence of RAFTs suggests that using stressors can be an effective approach to detecting flaky tests. Since high CPU load would slow down the machine the test is running on, other unrelated, time sensitive processes should not run on the same hardware simultaneously. A better approach would be to execute tests on a scheduled basis, when it is not used for other purposes or to have a dedicated machine that handles these tasks. The experiments in this study used a very capable machine and introduced an extreme CPU load. In practice, running the tests on an old or slow machine could be sufficient and would also be considered a more environmentally sustainable approach since it would remove the need to invest in new hardware. A downside of this approach would be the increased overhead and need for maintenance, as it would increase the risk of hardware failure and would make the group of machines more heterogenous. Using virtualization techniques like containerization could help mitigate the issue of excessive resource usage. Virtual machines, for example, can be configured with limited clock speeds to simulate the restricted CPU availability used in this study. Virtualization also allows multiple tasks to run simultaneously on a single physical machine without interference. Additionally, it addresses the heterogeneity problem, as each virtual machine can be configured identically.

Our findings from investigating RQ3 suggest that extracting test smells and code metrics from source code may provide insight into test flakiness, in line with current research [11, 12]. This implies the potential of statistical or learning-based approaches, possibly in combination with other features such as observed resource usage during a test execution, to determine whether a test failure is a flaky failure or not. Separating flaky failures from true failures is important in continuous integration projects because of reliance on regression testing to quickly detect and debug mistakes [25]. Instead of a developer having to needlessly debug a falsely flagged failure or rerunning the test case, an automated flakiness detector could save development teams much time. Furthermore, test smells and code metrics could be used as an indicator for developers by warning about threshold levels, even before a test case is executed. This could potentially reduce the amount of flakiness introduced to a project, preventing issues before they arise.

6.2 Method

In this section, we discuss the methodologies used by criticizing and justifying the choices we made during the study. The section provides considerations for future researchers to be able to reproduce these methods and improve upon them.

Investigating CPU usage over other resources

Expecting to find the most impactful results in CPU usage, in line with Silva et al. [6], we forewent other resources they investigated, such as memory and disk usage, to produce more CPU-related data. While this was clearly successful, due to the identified strong association between CPU usage and flakiness, other resources may have provided alternative associations that we now did not find. Furthermore, CPU usage might not be a direct cause of flakiness, but rather a symptom of a heavily loaded system. We hypothesize that the likely explanation of CPU-related flakiness is an increased number of context- and thread-switching, since the non-deterministic interleaving of processes and threads may cause non-deterministic behavior [35]. Interestingly, however, we initially did consider the number of context-switches per second as an alternative measurement to CPU usage, but when evaluated it performed much worse than CPU usage for explaining flakiness. But even though

CPU usage might not be the direct cause, an indirect cause may provide equal insight for tasks such as detecting or predicting resource-related flakiness.

Custom test execution framework

Because of our specific test environment (Section 4.1.2), and our need to collect the results within a couple of months, we created a custom test execution framework to manage resource monitoring, resource simulation, parallelization and output analysis. While other researchers cannot use our specific framework to reproduce our study, there was nothing special about the framework that prevents others from creating a similar one. We used open-source tools such as vmstat and stress-ng for resource monitoring and simulation, and typical Python statistics packages for data analysis. These tools are publicly available for everyone, allowing anyone to reproduce our study.

Industrial codebase

Investigating an industrial codebase prevents other researchers from reproducing our specific results, as we cannot publish the investigated test cases nor the SUT. Furthermore, there might be differences between the source code of industrial and open-source codebases that may affect code characteristics and flakiness, and therefore our results. For instance, Bavota et al. [31] found differences in the understanding of test maintenance between students and industrial developers, although comparing open-source developers to students might not be fair. Using an industrial codebase over an open-source one was an accepted tradeoff, as working with Ericsson gave us resources in terms of software, hardware and support necessary to perform our research. Without it, we could not have hoped to perform tens of thousands of test executions while stressing the hardware. We also believe that findings from an industrial codebase may provide unique insight into the research area of test flakiness. Previous research has shown varying results depending on the type of codebase studied. For example, Gruber et al. [27] examined flakiness in Python code and found that 59% of flaky tests were caused by Test Order Dependency. In contrast, Luo et al. [1], who investigated a mix of Java, C++, and Python code, found a drastically lower occurrence of Test Order Dependency, at 12%. Both authors investigated open-source codebases, and we argue that the large differences in flakiness between projects of different languages may very well apply to differences in open-source vs. industrial projects. For instance, industrial projects may be subject to stricter quality assurance, enforcing higher maintainability that could limit developer mistakes and the risk of introducing flakiness. By investigating an industrial codebase, unlike many other researchers, we contribute with findings necessary to gain a broader perspective on test flakiness.

Accuracy of RQ1 experiment

We consider our results to be valid due to the large sample size and the robustness of our statistical analysis. The results were tested for statistical significance using the Wilcoxon rank-sum test, which is suited for comparing non-parametric data distributions, such as those obtained from our test outcomes under varying CPU usage. Furthermore, part of our findings could be cross-validated against the findings of RQ2 and RQ3. In RQ2, all RIFTs from RQ1 were uniquely identified as being Concurrency-related flaky tests, supporting the groups of labels produced by RQ1. Similarly, RQ3 showed that all RIFTs had similar code characteristics, providing further support.

Relying on vmstat, which measures CPU usage directly based on kernel statistics, ensured that our CPU usage observations were valid. However, we only considered the average CPU usage over the entire execution of all tests, not individual test cases. This approach meant that temporary CPU spikes were not captured. Measuring individual test cases was infeasible

because we could not identify their start and end times within the test environment. All test cases were compiled into a single executable, preventing us from controlling the execution of individual cases without altering their source code. Despite this, the average CPU usage during all tests should be considered a reasonable approximation of individual test case CPU usage when applied to a large sample size. Since our experiment involved executing each investigated test case more than 20,000 times, we considered the sample large enough to approximate individual test case CPU usage with the average usage during all test cases.

Limited number of identified flaky tests

The limited number of identified flaky tests, i.e., eight RAFTs and three RIFTs, was an obstacle for answering RQ1. While we could statistically prove that computational resources impacts flakiness in some tests, we could not significantly claim the frequency of this happening. In order to fully explore the relation between RAFTs and RIFTs, one would have to analyze a larger dataset of more flaky tests.

Furthermore, the investigation of RQ3 was based on the same set of test cases as for RQ1, similarly preventing statistical analysis of test smells and code metrics in relation to RAFTs and RIFTs. Although this may have limited the validity of our results, we have provided support for our findings in this chapter through logical explanations, comparison with literature and discussion with subject matter experts. While applying our method to a larger dataset may yield additional findings, we consider the findings we have to be sound.

Selection of root causes

In both of the methods for RQ2, a list of root causes was used in order to categorize flaky test cases. The list can be found in Table 4.1 and contained 8 root causes based on the 14 root causes compiled by Parry et al. [4]. The 6 remaining root causes beyond the 8 used were: Randomness, Floating Point, Too Restrictive Range, Resource Leak, Network and Platform Dependency. They were thought not to be relevant as they had not appeared in an internal survey at the company. A larger selection of root causes could have lead to a more accurate categorization since there can be overlap between root causes and by having more alternatives, the survey recipients could potentially find a better label for the flakiness. It would also decrease the number of categorizations of Other.

For the fixes of flakiness, this probably did not affect the results significantly. There were only two commits classified as other, so even if they were classified as any other root cause, Async Wait and Concurrency would still have been chosen for further investigation. There were also sufficient cases of Async Wait and Concurrency to continue with the experiments. For the analysis of the test logs, additional root causes could have affected the classification of root causes, as discussed in Section 6.1.

Categorization of root causes

Given the shallow nature of the survey, some commits might have been labelled incorrectly. Only the author of the commits themselves investigated the commit and different authors could have interpreted the root causes in different ways. The authors were also not forced to leave additional notes for each classification. A more thorough approach would have been to ask more than one developer about the commit or to interview authors for more comprehensive explanations. Having more eyes on the commit would allow discrepancies to be highlighted for easier detection of misclassifications. Interviewing the authors would have provided more data to be used to make more informed decisions about interesting commits to investigate. The main issue with this would be the increased effort to participate for the developers. Answering a short survey about their own work takes a short amount of time while partaking in an interview or reviewing another developers work would be more time con-

suming. This could lead to a lower response rate and therefore a smaller dataset of commits to investigate.

In a similar way, the accuracy of the subject matter expert classification can also be challenged. However, the classifications were made by the individuals that had the most insight in the code and should therefore have the ability to make the most accurate estimations.

Selection of commits fixing flakiness

The selection of six commits from two project made during the course of 12 months is not sufficient to fully characterize the test flakiness in the projects. Instead, this study used a qualitative approach to establish similarities and differences in the root causes when put under varying CPU loads. If the rest of the commits of the chosen root causes were experimented on and analyzed, more discoveries could be made and more general conclusions could be drawn. The primary limitation for further exploration in this work was time, as the experiments required over a month of continuous execution. With additional time, another approach would be to run more executions on the same six commits to get even more reliable failure rate estimates.

The commits were not a random sample, taken from the two root causes chosen. They were chosen based on which component in the projects they affected, where similar components were preferred. A loose selection criteria was also that if two commits appeared to be very similar based on their description and affected files, only one was chosen in the experiments. This was done to create more variety in the commits.

There might have been more commits that aimed to fix flakiness in the codebase. The 63 commits were found using pattern matching of commit messages, described in Section 4.2.2. Commits could have used other wording to describe how it solved some sort of flakiness issue. There could also have been commits that implemented changes that indirectly affected the failure rate of one or more test cases. In this work, 63 commits were enough to be able to select six for further experimentation.

Test cases per execution

The number of test cases to run per execution was investigated to observe if it had an impact on the results in RQ2. Only one commit was chosen because experiments for entire suites are more time-consuming than running only one or two test cases. The choice of Async Wait Commit #1 was made before running the experiments. The only observed difference between running the entire suite or a single test case was the reduction in failures from 1 to 0 for low CPU. This could mean that running the suite introduced flaky behavior and that running only one test case would never lead to a failure. However, this could also be explained by a very low failure rate for either setup. For example, with a 0.058% failure rate, there would be a probability of about 50% that no failures occur when running 1200 executions. If the number of test cases per run had a significant impact, it would be more pronounced, so it appears to have had a minimal effect. If test order dependency had been chosen as a root cause for the commit selection, then the number of test cases run would have had more impact. This is because the test cases chosen could be dependent on other test cases in the same suite.

It is difficult to say if the CPU loads chosen for the experiments had a large impact on the results. The low CPU loads did not lead to high failure rate in any test case, so choosing other CPU loads in the same range would not have affected the outcomes significantly. Using CPU loads closer to 80% could have lead to more failures, but it would also lead to more executions being excluded from the final results since the CPU usage of the executions and the CPU usage of the stressor processes could exceed 80%. Regarding the high CPU load, increasing the load might have led to higher failure rates for the test cases. This could have affected some of the results such as for the test cases with 0.08% failure rate for high CPU usage. However,

a CPU load of above 80% might happen infrequently enough in a production environment that even higher CPU loads would not be interesting to investigate.

The exclusion of executions with low CPU loads exceeding 80% CPU usage should not be viewed as manipulation of the results. This is because the aim of the experiments was to investigate how CPU usage relates to flakiness and not to investigate how CPU load affects CPU usage. Furthermore, this exclusion was only used in the early stages of the experiments while adjusting the level of parallelization to avoid exceeding the 80% limit.

Heuristic test smell detection rules

Due to limited time and experience in creating the C++ parsing tool, and because of project-specific details, Eager Test, Lazy Test and Test Code Duplication were assigned heuristic detection rules.

Eager Test, defined by a test method calling multiple production methods [13], had to be reinterpreted as test methods did not directly call production methods. In the test environment, described in Section 4.1.2, a test case asynchronously triggered behaviors of the SUT through API calls. Each triggered behavior was separated by a synchronization directive (Listing 4.3), such that multiple behaviors in a test could be measured by the number of synchronization points. Eager Test was then detected as a test with multiple synchronization points, implying multiple tested SUT behaviors.

Lazy Test is originally defined by multiple tests calling the same production method [13], and had to be reinterpreted for the same reason as Eager Test. Our rule, checking if two tests perform identical operations, effectively captured tests exercising the same SUT behavior. However, due to the strictness of the rule, the identified smells were an underestimation, as tests could be lazy while performing slightly different operations.

Test Code Duplication refers to duplicated test case code, but no specific threshold is mentioned in literature [13]. We selected the heuristic detection rule of two tests being 80% or more similar with regards to effective operations, which captures if two tests are directly or indirectly performing similar operations. The threshold of 80% was found to yield the highest variance between non-flaky tests, RAFTs and RIFTs after experimentation on our dataset, making it a project-specific parameter. The ambiguity in the definition of the smell made it difficult to select an exact rule and threshold, but this likely also affects other detection tools, making the problem non-specific to our research.

Although these rules are heuristic, we consider them to effectively capture the essential meanings of the original smells described by Van Deursen et al. [13], as explained above, making the smells identified using these rules valid. Because of the described heuristic detection, the absolute frequencies of the smells may not be comparable to other research. However, the distributions of smells across flaky and non-flaky tests should be comparable. For instance, if flaky tests indeed tend to exhibit Eager Test more often, then multiple different valid rules detecting Eager Test should reveal similar trends.

Furthermore, because the rules are specific to a unique test environment, they cannot be directly applied to all projects. To reproduce our study, one would have to invent rules that similarly capture the essential meaning of the original test smells.

Source criticism

It is important to note the differences between peer-reviewed sources and more industry-focused, non-peer-reviewed sources and how they are used in this thesis. Sources such as conference papers, technical reports, and industry whitepapers can provide useful context but may not undergo the same level of scrutiny as peer-reviewed journal articles. In this study, mainly peer-reviewed journals are used to establish previous research in the area of flaky tests. Similarly, a significant portion of the background makes use of peer-reviewed sources such as IEEE and ACM. These are considered to be reputable sources that engage in

thorough peer-review practices. However, sources such as Fowler [25] and Amazon [20] have not been peer-reviewed. To ensure their reliability, we assessed the sources' authenticity and the authors' other work. For Fowler [25], the author's book Refactoring [58] is widely cited and highly regarded, serving as a base for the work of our other peer-reviewed sources such as Van Deursen et al. [13]. Some relatively recent peer-reviewed papers with few citations have also been used in this work which could pose a threat to credibility. However, these works are used alongside peer-reviewed works to complement the arguments made.

6.3 The work in a wider context

Although not directly investigated by this thesis, the environmental impact of automated software testing is important to consider. Running automated software comes with energy costs most likely not observed by developers. When a flaky test fails, the developer often rely on rerunning the test suite in order to commit their changes [25, 47], and therefore cause needless energy consumption. Our research, aimed at helping developers to identify flakiness, can potentially be used to reduce the number of flaky tests in a project, and therefore eliminate flakiness-related energy consumption.

However, our proposed method of identifying flakiness by rerunning tests a large number of times does itself likely consume much energy. Just the experiment for RQ1 executed our set of tests over 20,000 times each. Furthermore, simulating CPU usage while doing so drastically increased the energy consumption. Whether the long term reduced energy costs because of reduced flakiness outweighs the cost of identifying them is difficult to say, making the environmental impact of our research hard to determine. One argument for its benefit is that companies and projects with large-scale testing may have increased use, as for instance Luo et al. [1] mentions that Google at one point had 73,000 test failures due to flakiness just in the period of 15 months.

Furthermore, the presence of flaky tests leads to frustration among developers [25]. Flaky tests undermine confidence in the test suite, forcing developers to waste time investigating whether a failure is caused by recent changes or due to flakiness. If this becomes a repeated pattern, it could over time lead to stress and developer burnout. By using our research to reduce flakiness, development teams could potentially eliminate these factors, improving mental health conditions and morale.

Finally, reliable software testing has an impact on society, as it allows the development of robust and secure systems globally. For instance, the CrowdStrike incident caused global outage in Microsoft Windows computers [59], and was caused by an issue missed in testing [60]. This highlights the importance of software testing, and the benefits of research in the area. Hopefully, our research provides insight into the hazard of test flakiness, and specifically resource-affected flakiness, to enable more robust software testing.

Summary: This chapter presented an analysis of the results and methods used in this work. It discussed practical applications of the findings in addition to key insights about root causes, test code characteristics and resource impact. The chapter also examined the reliability and replicability of the methods conducted, along with an evaluation of the sources used. Finally, it explored ethical and societal implications of the work. The next chapter, Conclusion, will provide the answers to the research questions and explore different evolutions, enhancements and future pathways for further research.

---

7 Conclusion

In this chapter we answer each research question proposed in Section 1.3, and identify opportunities for further exploration in this area of research. Each research question is answered separately and the chapter concludes with a section covering future work.

7.1 Overview of results

The primary focus of our research was to explore the relationship between computational resource availability and test flakiness in an industrial software testing environment. Test flakiness, defined by random test failures without changes in the code under test, is a hazard for regression testing. We aimed to identify the impact of system resource limitations on flaky tests, analyze root causes of flakiness in this context, and evaluate the predictive potential of test smells and code metrics. By conducting extensive experiments and qualitative analysis in an industrial context, our research contributes to understanding how resource availability and source code characteristics impact software testing reliability.

RQ1: How does computational resource availability correlate with test flakiness?

The objective of RQ1 was to explore the correlation between computational resource availability and test flakiness in an industrial codebase. By conducting an experiment of rerunning 199 test cases under simulated CPU usage, we identified 11 flaky cases and found significant positive relations between system CPU usage and flakiness in 8 of them. These findings suggest an impact of resource availability on test flakiness in some, but not all tests.

RQ2: How do specific root causes of flaky tests and computational resource availability contribute to flakiness?

The experiments of RQ2 uncovered some general trends regarding how computational resource availability affected test flakiness root causes, especially Async Wait and Concurrency. Async Wait was found to be resource-affected in every instance observed. Test cases with the Concurrency test flakiness root cause were found to have more varied results where some

test cases were resource-independent and the resource-affected cases exhibited substantial variation in failure rates.

RQ3: How do test smells, code metrics and computational resource availability predict test flakiness?

The goal of RQ3 was to investigate the potential of test smells and code metrics found in test case source code as predictors of resource-affected test flakiness. Our results and qualitative analysis suggest promise in the use of Sleepy Test and Conditional Test logic as weak predictors of resource-affected flakiness, while Eager Test, Lazy Test and Test Code Duplication could be used to differentiate between resource-affected and resource-independent flakiness. The code metrics Lines of Code and Cyclomatic Complexity, both measuring maintainability, perform well when the source code of test helper functions is considered. Metrics counting thread sleeps and project-specific operations, on the other hand, provide little information gain beyond maintainability.

7.2 Future work

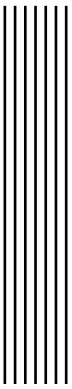
There are several directions of research that would be useful to elaborate upon. For one, the scope of the experiments presented in the study could be widened to incorporate more projects, more root causes and more test smells and code metrics.

Applying the experiments provided in this study to more projects could contribute to a concrete knowledge base to derive more scientific and practical work from. Differentiating RAFTs and RIFTs in different industries, applications and programming languages would lead to a better understanding of the nature of flaky tests, and a better understanding can consequently lead to better prevention of non-determinism.

Further exploration of root causes for RAFTs and RIFTs could also prove fruitful. Async Wait and Concurrency became the focus of the research about root causes in this study. Additional investigation into these root causes could uncover more general trends of how they affect resource dependency for flaky tests. Supplementary root causes could be interesting as well. Test Order Dependency and Too Restrictive Range were discussed in this study and could act as a valuable starting point. Qualitative research methods on underlying reasons for root causes to be resource-affected or not would also be beneficial to the work in this area.

Regarding the usage of test smells for predicting resource-affected flakiness, we see the potential of exploring a larger set of test smells. Among the original smells by Van Deursen et al. [13], we excluded Mystery Guest, Resource Optimism and Test Run War because these are based on the use of external resources such as files and databases that could not be used by developers in the investigated test environment. These test smells could be relevant to flakiness in other projects, as the use of external resources is a potential source of flakiness [25, 1]. We hypothesize that they can specifically relate to resource-affected flakiness, as external resources may become unavailable when computational resource availability is limited. Other smells related to production code, such as Indirect Testing and For Testers Only, were also excluded as we could not analyze relations between test and production code. Including production code-based test smells could provide alternative insights to our findings, allowing test flakiness prediction based on both test and production code.

Practical advancements from the basis of this work should not be overlooked. Using resource constraints to induce flakiness could be developed further to not only detect flakiness but also classify the type of flakiness that tests exhibit. This could provide an effective entrypoint for debugging flakiness for developers. Analyzing test suites by limiting resource availability could provide a solid basis for advanced testing strategies, such as queuing critical testing tasks to run only when sufficient resources can be provided.



Bibliography

[1] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. "An empirical analysis of flaky tests". In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (fse-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 643–653. DOI: 10.1145/2635868.2635920.

[2] Martin Gruber and Gordon Fraser. "A survey on how test flakiness affects developers and what support they need to address it". In: *15th IEEE conference on software testing, verification and validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022, pp. 82–92. DOI: 10.1109/ICST53961.2022.00020.

[3] Md Tajmilur Rahman and Peter C. Rigby. "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds". In: *Proceedings of the 2018 ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 857–862. DOI: 10.1145/3236024.3275529.

[4] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. "A survey of flaky tests". In: *ACM trans. softw. eng. methodol.* 31.1 (2022), 17:1–17:74. DOI: 10.1145/3476105.

[5] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. "The effects of computational resources on flaky tests". In: *IEEE trans. software eng.* 50.12 (2024), pp. 3104–3121. DOI: 10.1109/TSE.2024.3462251.

[6] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. "Shake it! Detecting flaky tests caused by concurrency with Shaker". In: *IEEE international conference on software maintenance and evolution, ICSME 2020, Adelaide, Australia, September 28 - october 2, 2020*. IEEE, 2020, pp. 301–311. DOI: 10.1109/ICSME46990.2020.00037.

[7] Gillian Ann Yost. "Finding flaky tests in JavaScript applications using stress and test suite reordering". PhD thesis. The University of Texas at Austin, 2023.

[8] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. "Understanding flaky tests: the developer's perspective". In: *Proceedings of the ACM joint meeting on European software engineering conference and symposium on the foundations of software*

engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 830–840. DOI: 10.1145/3338906.3338945.

[9] Wing Lam, Kivanç Muslu, Hitesh Sajnani, and Suresh Thummalapenta. "A study on the lifecycle of flaky tests". In: *ICSE '20: 42nd international conference on software engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 1471–1482. DOI: 10.1145/3377811.3381749.

[10] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. "An empirical study of flaky tests in JavaScript". In: *IEEE international conference on software maintenance and evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 24–34. DOI: 10.1109/ICSME55016.2022.00011.

[11] Bruno Henrique Pachulski Camara, Marco Silva, André Takeshi Endo, and Silvia R. Vergilio. "On the use of test smells for prediction of flaky tests". In: *SAST'21: Brazilian symposium on systematic and automated software testing, Joinville, Brazil, 27 September 2021 - 1 October 2021*. Ed. by Cristiano D. Vasconcellos, Karina Girardi Roggia, Paulo Bousfield, Vanessa Collere, Marcelo Eler, and Wesley K. G. Assunção. ACM, 2021, pp. 46–54. DOI: 10.1145/3482909.3482916.

[12] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. "Toward static test flakiness prediction: a feasibility study". In: *MaLTeSQuE@ESEC/SIGSOFT FSE 2021: proceedings of the 5th international workshop on machine learning techniques for software quality evolution, Athens, Greece, 23 August 2021*. Ed. by Apostolos Ampatzoglou, Daniel Feitosa, Gemma Catolino, and Valentina Lenarduzzi. ACM, 2021, pp. 19–24. DOI: 10.1145/3472674.3473981.

[13] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. "Refactoring test code". In: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer. 2001, pp. 92–95.

[14] IBM. *What is software testing?* 2024. URL: https://www.ibm.com/topics/software-testing (visited on 2024-11-03).

[15] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. *JUnit 5 user guide*. URL: https://junit.org/junit5/docs/current/user-guide/ (visited on 2024-11-03).

[16] GoogleTest. *GoogleTest user's guide*. URL: http://google.github.io/googletest/ (visited on 2024-11-03).

[17] pytest. *pytest documentation*. URL: https://docs.pytest.org/en/stable/index.html (visited on 2024-11-03).

[18] J.A. Whittaker. "What is software testing? And why is it so hard?" In: *IEEE software* 17.1 (2000), pp. 70–79.

[19] Hareton K. N. Leung and Lee J. White. "A study of integration testing and software regression at the integration level". In: *Proceedings of the conference on software maintenance, ICSM 1990, San Diego, CA, USA, 26-29 November, 1990*. IEEE, 1990, pp. 290–301. DOI: 10.1109/ICSM.1990.131377.

[20] AWS Amazon. *What is unit testing? - Unit testing explained - AWS*. URL: https://aws.amazon.com/what-is/unit-testing/ (visited on 2024-10-27).

[21] Michael Ellims, James Bridges, and Darrel C. Ince. "The economics of unit testing". In: *Empir. softw. eng.* 11.1 (2006), pp. 5–31. DOI: 10.1007/S10664-006-5964-9.

[22] Lionel C. Briand and Yvan Labiche. "A UML-based approach to system testing". In: *Softw. syst. model.* 1.1 (2002), pp. 10–42. DOI: 10.1007/S10270-002-0004-8.

[23] Simeon C. Ntafos. "A comparison of some structural testing strategies". In: *IEEE trans. software eng.* 14.6 (1988), pp. 868–874. DOI: 10.1109/32.6165.

[24] "IEEE standard for software and system test documentation". In: *IEEE std 829-2008* (2008), pp. 1–150. DOI: 10.1109/IEEESTD.2008.4578383.

[25] Martin Fowler. *Eradicating non-determinism in tests*. 2011. URL: https://martinfowler.com/articles/nonDeterminism.html (visited on 2024-10-02).

[26] Sebastian G. Elbaum, Gregg Rothermel, and John Penix. "Techniques for improving regression testing in continuous integration development environments". In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 235–245. DOI: 10.1145/2635868.2635910.

[27] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. "An empirical study of flaky tests in Python". In: *14th IEEE conference on software testing, verification and validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 2021, pp. 148–158. DOI: 10.1109/ICST49551.2021.00026.

[28] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. "Test smell detection tools: A systematic mapping study". In: *EASE 2021: evaluation and assessment in software engineering, Trondheim, Norway, June 21-24, 2021*. Ed. by Ruzanna Chitchyan, Jingyue Li, Barbara Weber, and Tao Yue. ACM, 2021, pp. 170–180. DOI: 10.1145/3463274.3463335.

[29] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. "tsDetect: an open source test smells detection tool". In: *ESEC/FSE '20: 28th ACM joint European software engineering conference and symposium on the foundations of software engineering, virtual event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 1650–1654. DOI: 10.1145/3368089.3417921.

[30] Gerard Meszaros. *xUnit test patterns*. 2011. URL: http://xunitpatterns.com/index.html (visited on 2024-11-01).

[31] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave W. Binkley. "Are test smells really harmful? An empirical study". In: *Empir. softw. eng.* 20.4 (2015), pp. 1052–1094. DOI: 10.1007/S10664-014-9313-0.

[32] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. "iFixFlakies: a framework for automatically fixing order-dependent flaky tests". In: *Proceedings of the ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 545–555. DOI: 10.1145/3338906.3338925.

[33] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[34] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006. (Visited on 2024-10-02).

[35] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[36] Andrew S. Tanenbaum. *Modern operating systems, 3rd edition*. Pearson Prentice-Hall, 2009. ISBN: 0138134596. URL: https://www.worldcat.org/oclc/254320777.

[37] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *Proceedings of the 13th international conference on architectural support for programming languages and operating systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. Ed. by Susan J. Eggers and James R. Larus. ACM, 2008, pp. 329–339. DOI: 10.1145/1346281.1346323.

[38] Red Hat. *Understanding and mitigating the dirty COW vulnerability*. 2016-11. URL: https://www.redhat.com/en/blog/understanding-and-mitigating-dirty-cow-vulnerability (visited on 2024-10-25).

[39] Manuel Breugelmans and Bart Van Rompaey. "TestQ: exploring structural and maintenance characteristics of unit test suites". In: *WASDeTT-1: 1st international workshop on advanced software development tools and techniques*. Citeseer. 2008, p. 11.

[40] Michaela Greiler, Arie van Deursen, and Margaret-Anne D. Storey. "Automated detection of test fixture strategies and smells". In: *Sixth IEEE international conference on software testing, verification and validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 322–331. DOI: 10.1109/ICST.2013.45.

[41] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne D. Storey. "Strategies for avoiding text fixture smells during software evolution". In: *Proceedings of the 10th working conference on mining software repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim. IEEE Computer Society, 2013, pp. 387–396. DOI: 10.1109/MSR.2013.6624053.

[42] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. "Automatic test smell detection using information retrieval techniques". In: *2018 IEEE international conference on software maintenance and evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 311–322. DOI: 10.1109/ICSME.2018.00040.

[43] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. "SoCRATES: Scala radar for test smells". In: *Proceedings of the tenth ACM SIGPLAN symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*. Ed. by Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom. ACM, 2019, pp. 22–26. DOI: 10.1145/3337932.3338815.

[44] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. "Just-in-time test smell detection and refactoring: the DARTS project". In: *ICPC '20: 28th international conference on program comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 441–445. DOI: 10.1145/3387904.3389296.

[45] Railana Santana, Luana Almeida Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor A. X. Costa, and Ivan Machado. "RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring". In: *34th brazilian symposium on software engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*. Ed. by Everton Cavalcante, Francisco Dantas, and Thaís Batista. ACM, 2020, pp. 374–379. DOI: 10.1145/3422392.3422510.

[46] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor A. X. Costa, and Ivan Machado. "On the influence of test smells on test coverage". In: *Proceedings of the XXXIII Brazilian symposium on software engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*. Ed. by Ivan do Carmo Machado, Rodrigo Rocha Gomes e Souza, Rita Suzana Pitangueira Maciel, and Cláudio Sant'Anna. ACM, 2019, pp. 467–471. DOI: 10.1145/3350768.3350775.

[47] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. "Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects". In: *31st IEEE international symposium on software reliability engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*. Ed. by Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng. IEEE, 2020, pp. 403–413. DOI: 10.1109/ISSRE5003.2020.00045.

[48] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. "DeFlaker: automatically detecting flaky tests". In: *Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman. ACM, 2018, pp. 433–444. DOI: 10.1145/3180155.3180164.

[49] John Micco. "The state of continuous integration testing @Google". In: *ICST*. 2017.

[50] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. "Empirically revisiting the test independence assumption". In: *International symposium on software testing and analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. Ed. by Corina S. Pasareanu and Darko Marinov. ACM, 2014, pp. 385–396. DOI: 10.1145/2610384.2610404.

[51] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. "iDFlakies: a framework for detecting and partially classifying flaky tests". In: *12th IEEE conference on software testing, validation and verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 312–322. DOI: 10.1109/ICST.2019.00038.

[52] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. "Detecting assumptions on deterministic implementations of non-deterministic specifications". In: *2016 IEEE international conference on software testing, verification and validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 80–90. DOI: 10.1109/ICST.2016.40.

[53] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. "NonDex: a tool for detecting and debugging wrong assumptions on Java API specifications". In: *Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. Ed. by Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su. ACM, 2016, pp. 993–997. DOI: 10.1145/2950290.2983932.

[54] Matias Waterloo, Suzette Person, and Sebastian G. Elbaum. "Test analysis: searching for faults in tests (N)". In: *30th IEEE/ACM international conference on automated software engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 149–154. DOI: 10.1109/ASE.2015.37.

[55] Gunnar Blom. *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 1980.

[56] Thomas J McCabe. "A complexity measure". In: *IEEE transactions on software engineering* 4 (1976), pp. 308–320.

[57] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. "On the distribution of test smells in open source Android applications: an exploratory study". In: *Proceedings of the 29th annual international conference on computer science and software engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*. Ed. by Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel. ACM, 2019, pp. 193–202. DOI: 10.5555/3370272.3370293.

[58] Martin Fowler. *Refactoring - improving the design of existing code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7. URL: http://martinfowler.com/books/refactoring.html.

[59] CISA. *Widespread IT outage due to CrowdStrike update*. 2024-07. URL: https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update (visited on 2024-12-22).

[60] CrowdStrike. *External technical root cause analysis — Channel File 291*. 2024-08. URL: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (visited on 2024-12-22).

A Figure of Experiment Test Execution Frequency for RQ1

This appendix describes the number of runs per test case from the experiment in RQ1. The figure is referenced in Section 5.1.1 and describes how many times each test case was executed, sorted by number of executions. The figure visualizes two effects: the excluded test suites have less executions (the right-most 25 test cases in the figure) and later test cases in order of execution have less executions due to immediate exit when earlier test cases fail.

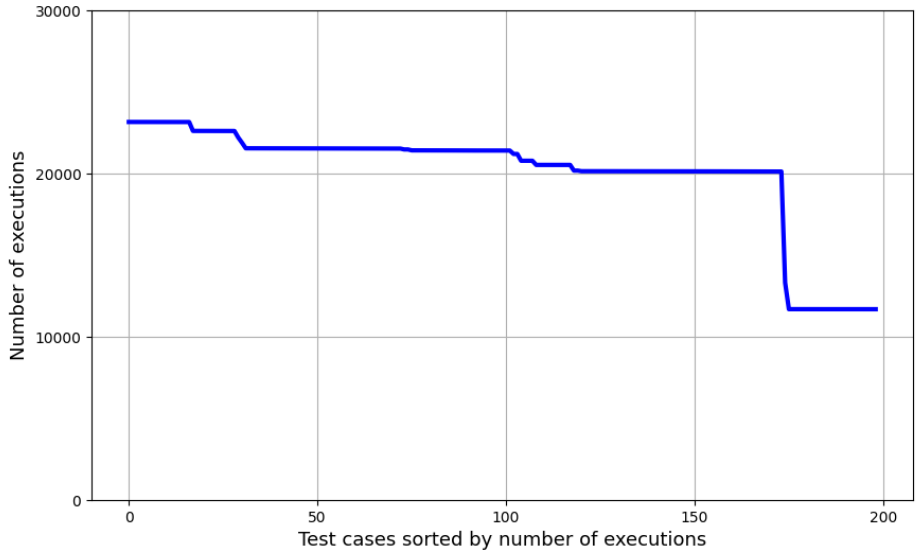


Figure A.1: Number of executions per test case

B Figures of Experiment Output for RQ1

This appendix contains figures detailing output of the experiment performed for RQ1. Each investigated test case is plotted with its failure rate over intervals of CPU usage.

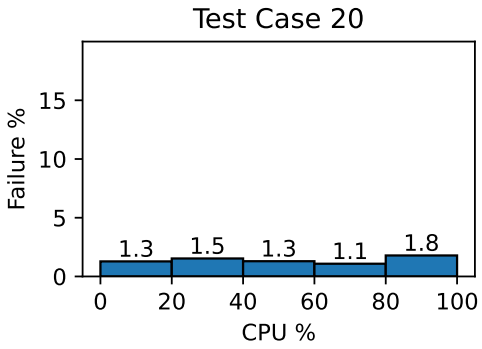


Figure B.1: Test Case 20, failure rate over CPU usage

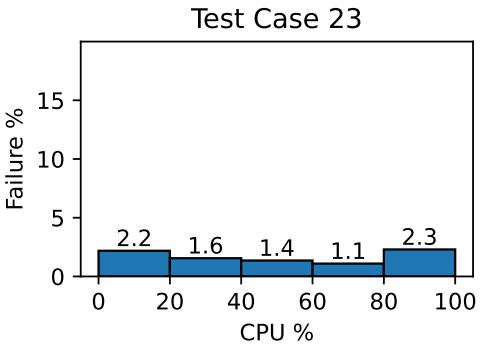


Figure B.2: Test Case 23, failure rate over CPU usage

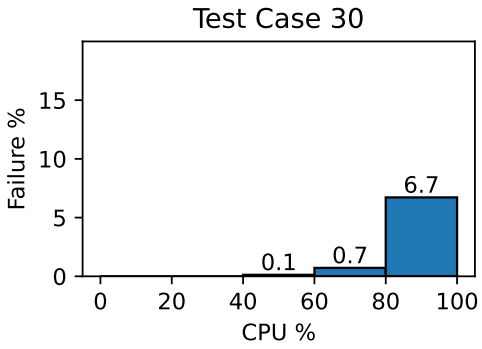


Figure B.3: Test Case 30, failure rate over CPU usage

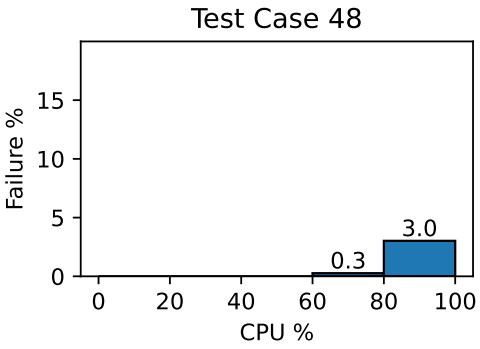


Figure B.4: Test Case 48, failure rate over CPU usage

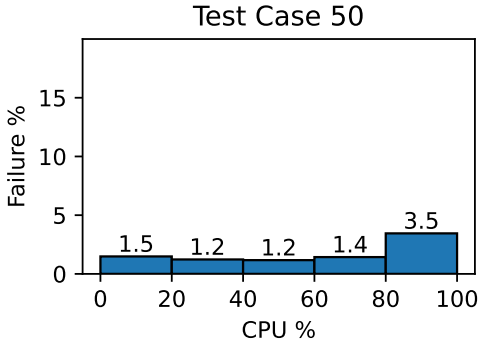


Figure B.5: Test Case 50, failure rate over CPU usage

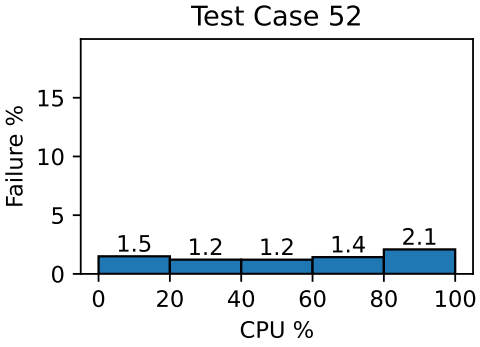


Figure B.6: Test Case 52, failure rate over CPU usage

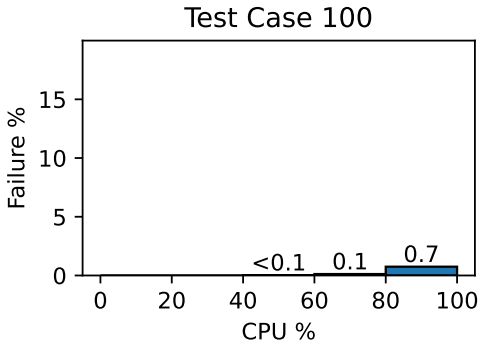


Figure B.7: Test Case 100, failure rate over CPU usage

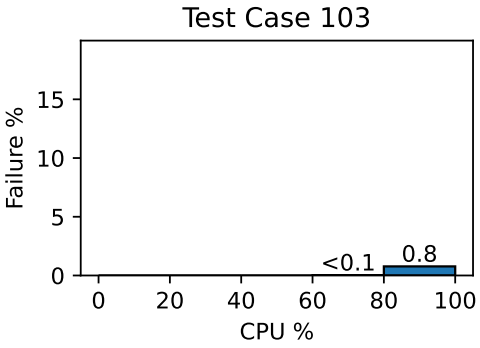


Figure B.8: Test Case 103, failure rate over CPU usage

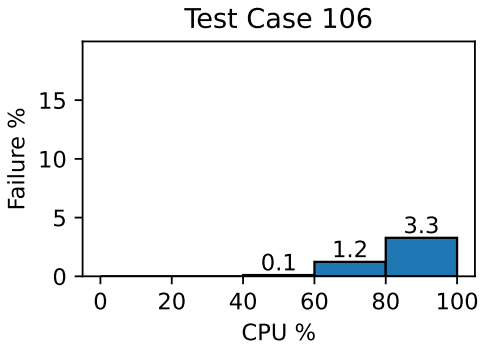


Figure B.9: Test Case 106, failure rate over CPU usage

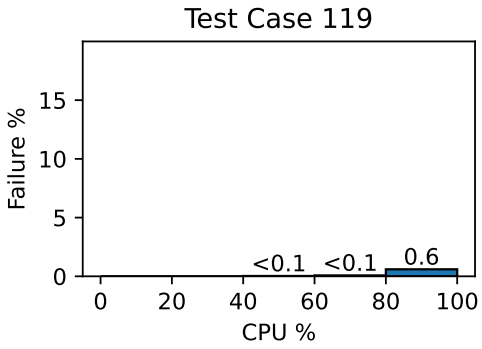


Figure B.10: Test Case 119, failure rate over CPU usage

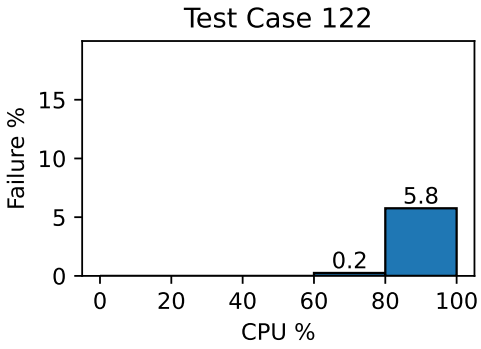


Figure B.11: Test Case 122, failure rate over CPU usage

C Figures of Impact of Parallelism on RQ1

This appendix contains figures detailing the impact of parallel test execution on the results of RQ1. Each investigated test case is plotted with its failure rate over intervals of CPU usage, split into subplots of different levels of parallelism. The subplots labelled "unfiltered" show the failure rates of all executions, without filtering by a specific level of parallelism.

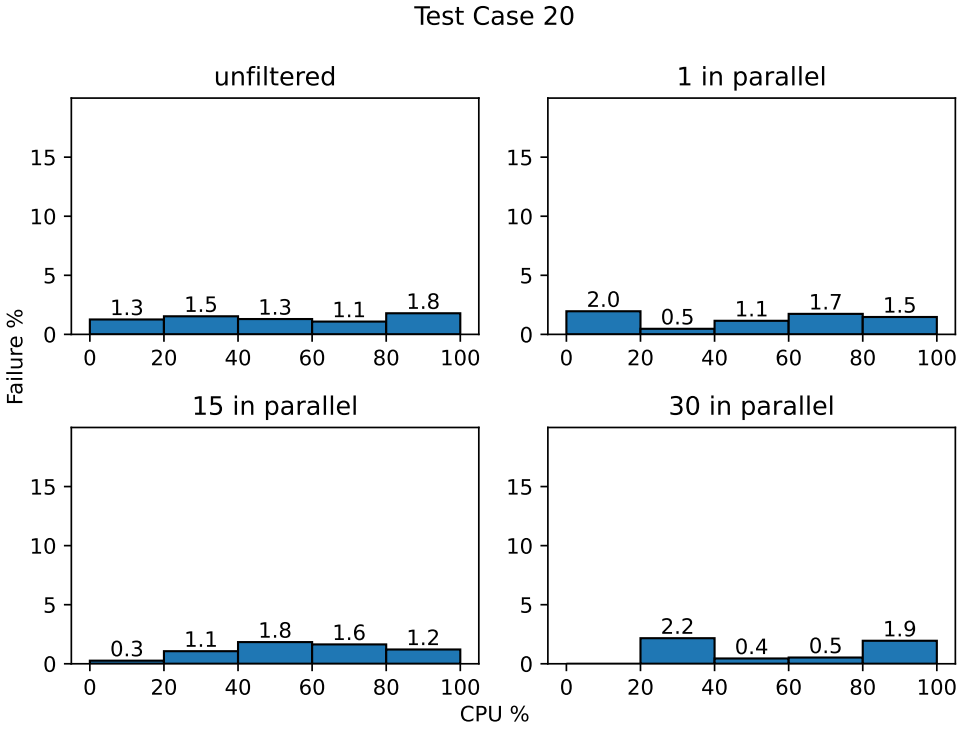


Figure C.1: Test Case 20, failure rate over CPU usage, split by no. parallel executions

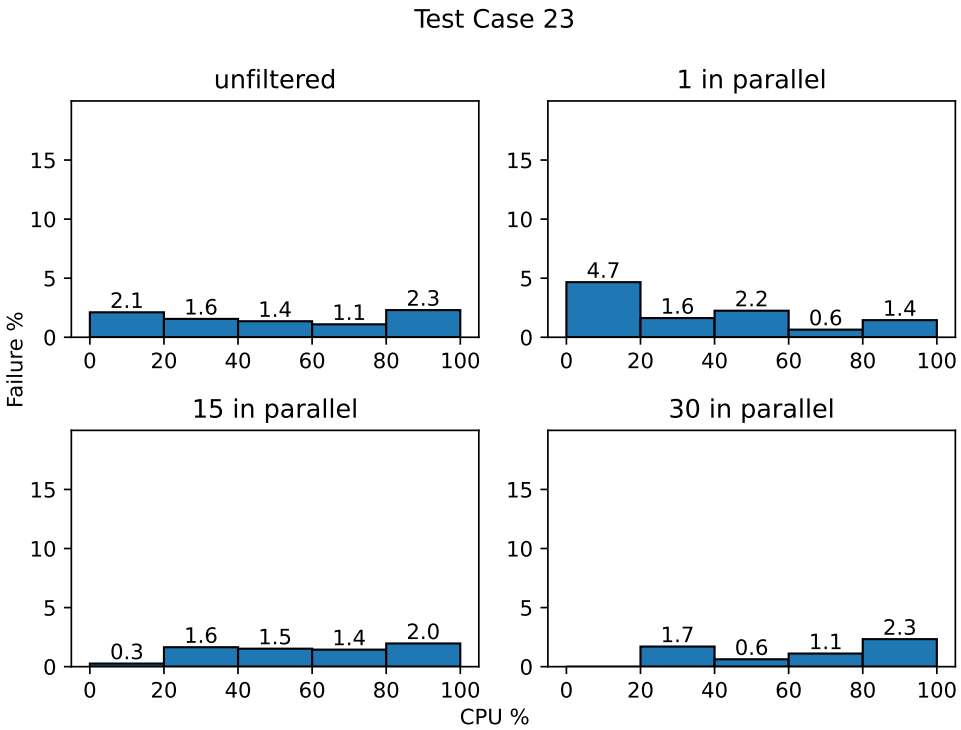


Figure C.2: Test Case 23, failure rate over CPU usage, split by no. parallel executions

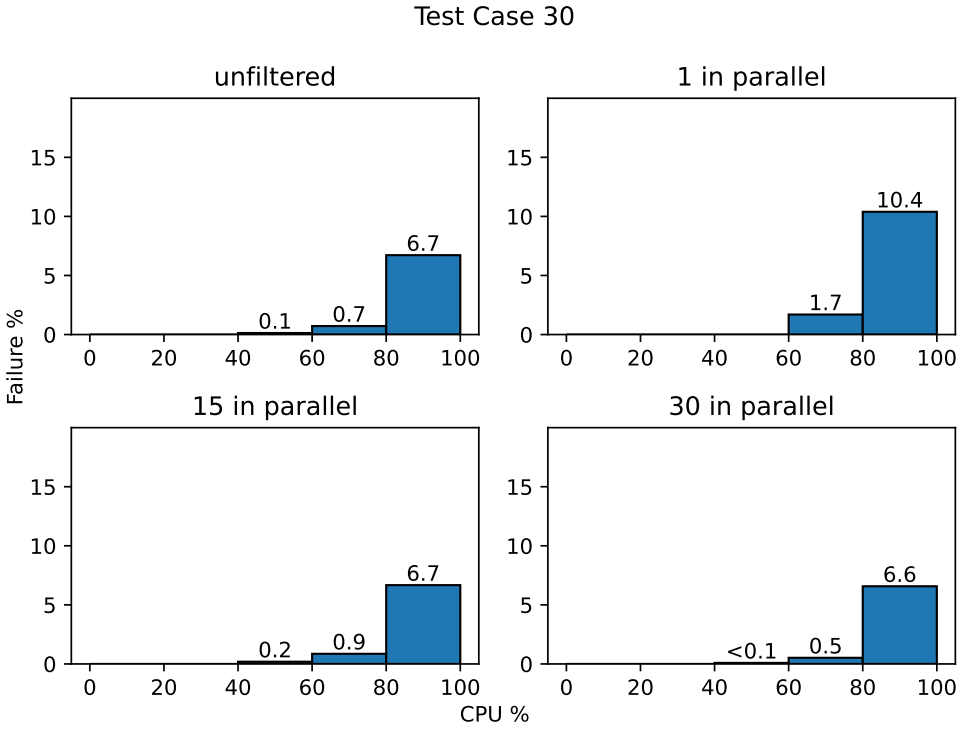


Figure C.3: Test Case 30, failure rate over CPU usage, split by no. parallel executions

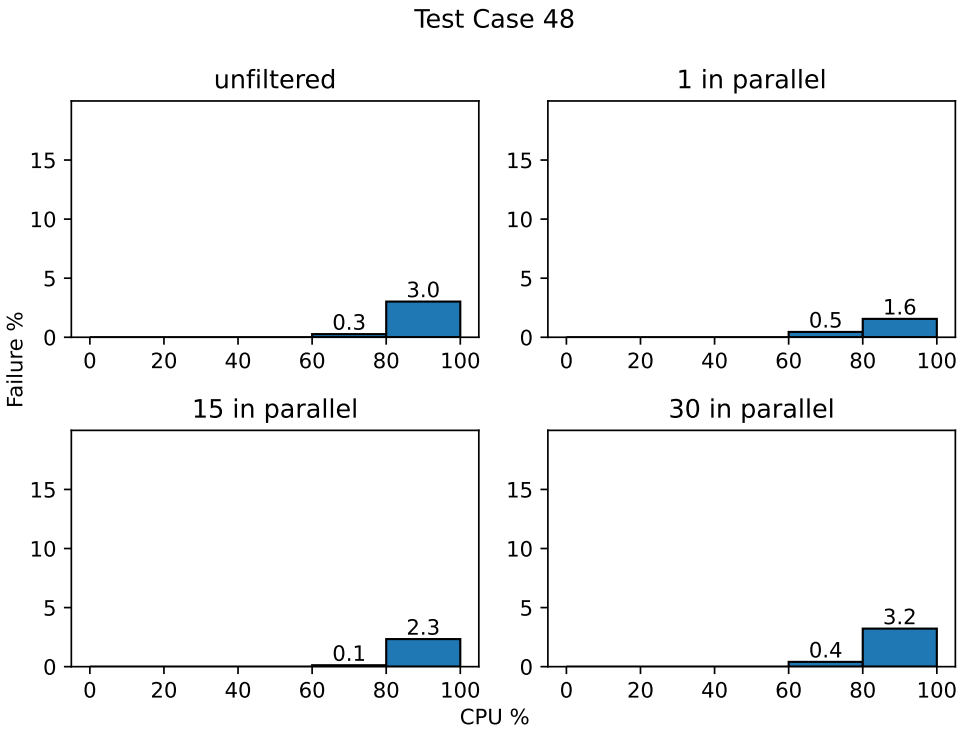


Figure C.4: Test Case 48, failure rate over CPU usage, split by no. parallel executions

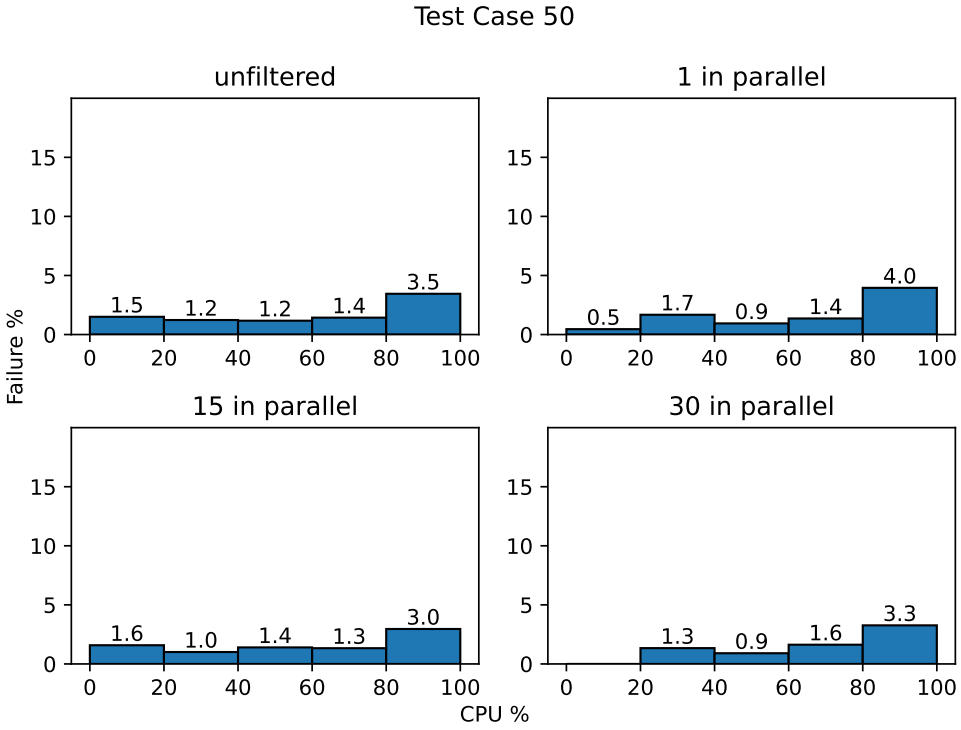


Figure C.5: Test Case 50, failure rate over CPU usage, split by no. parallel executions

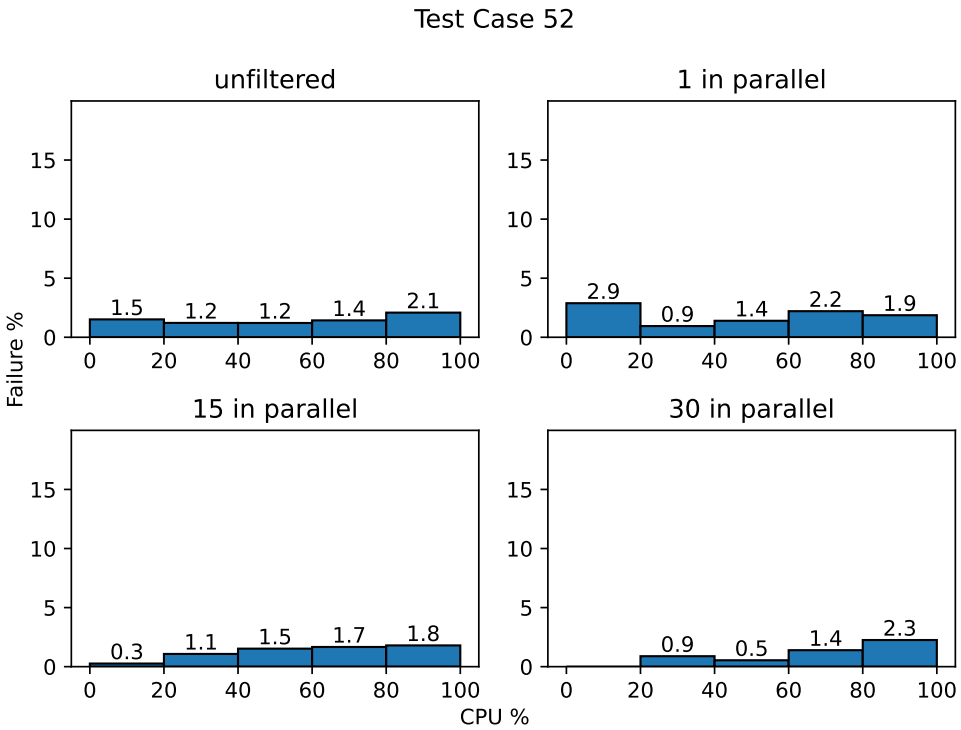


Figure C.6: Test Case 52, failure rate over CPU usage, split by no. parallel executions

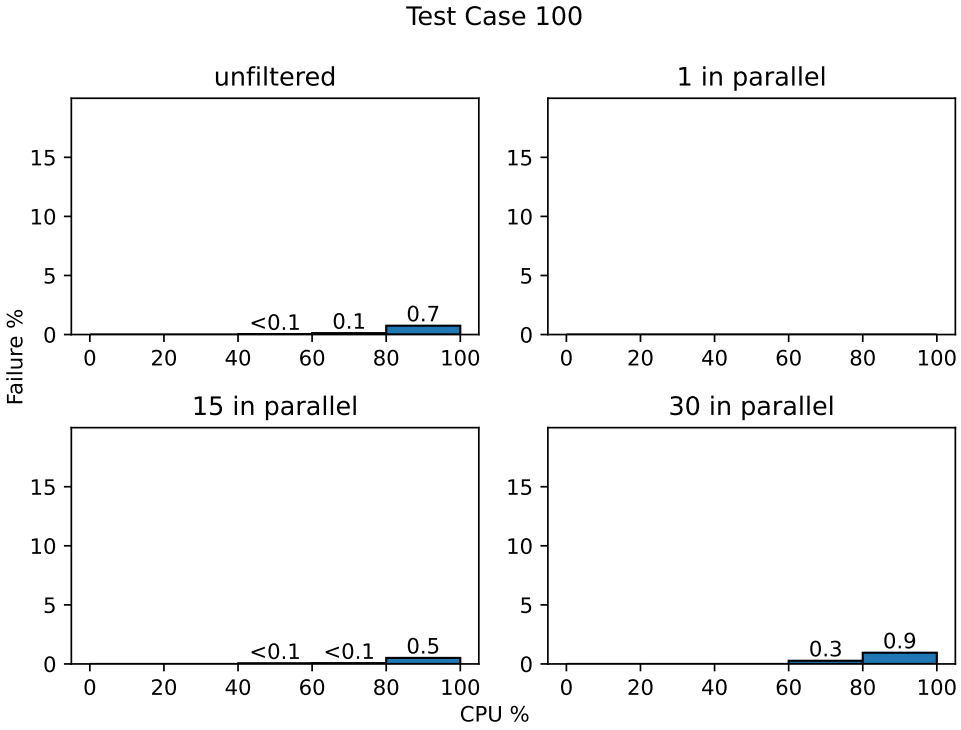


Figure C.7: Test Case 100, failure rate over CPU usage, split by no. parallel executions

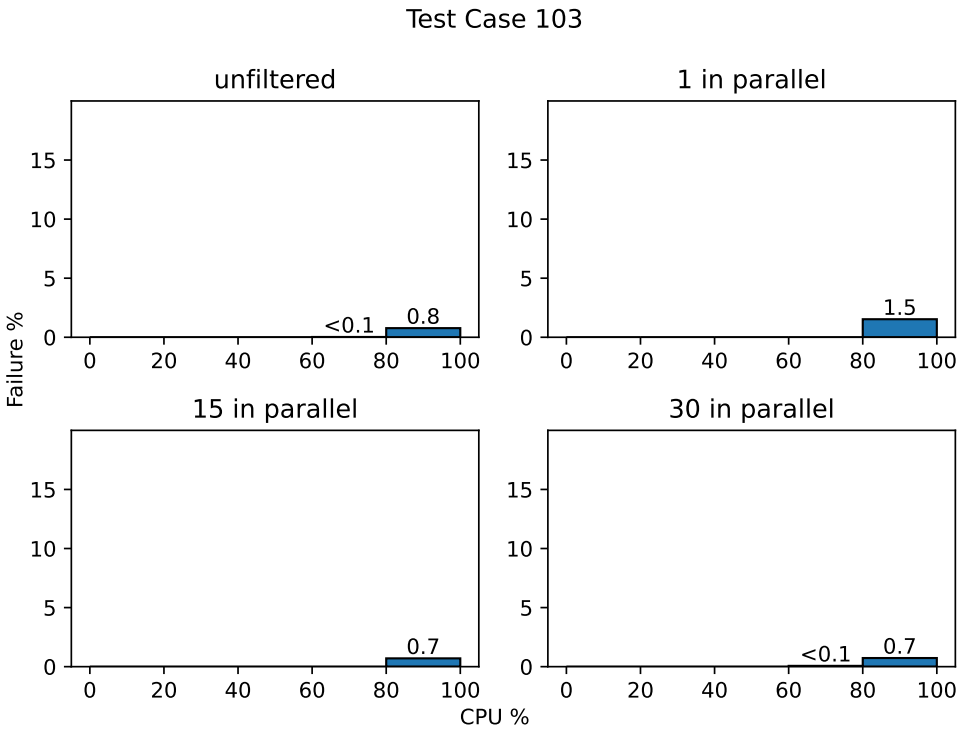


Figure C.8: Test Case 103, failure rate over CPU usage, split by no. parallel executions

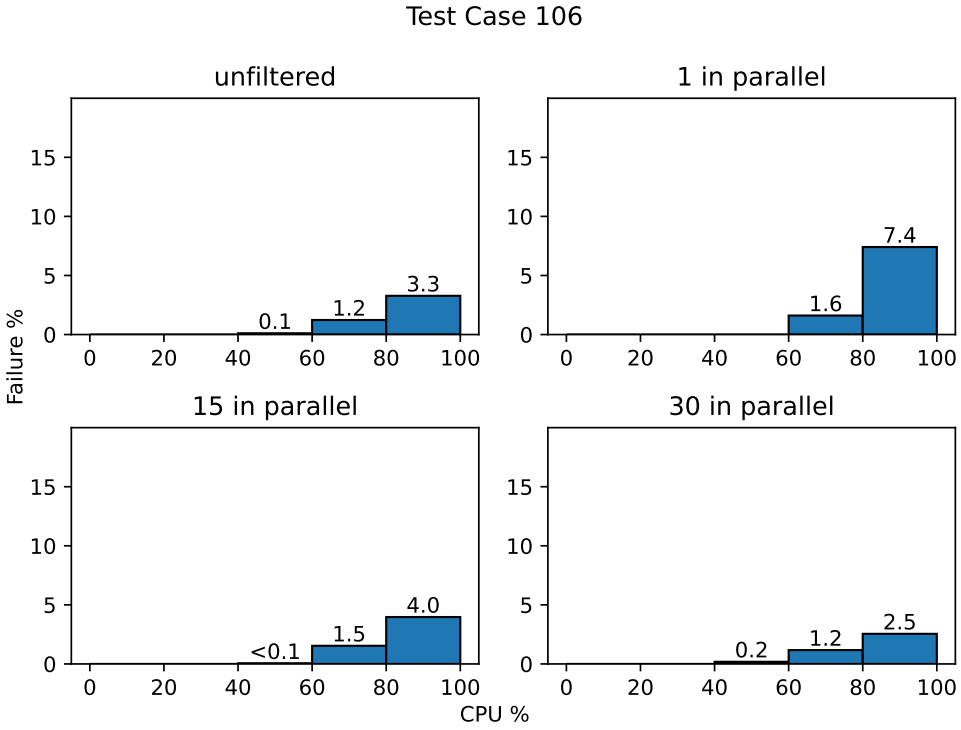


Figure C.9: Test Case 106, failure rate over CPU usage, split by no. parallel executions

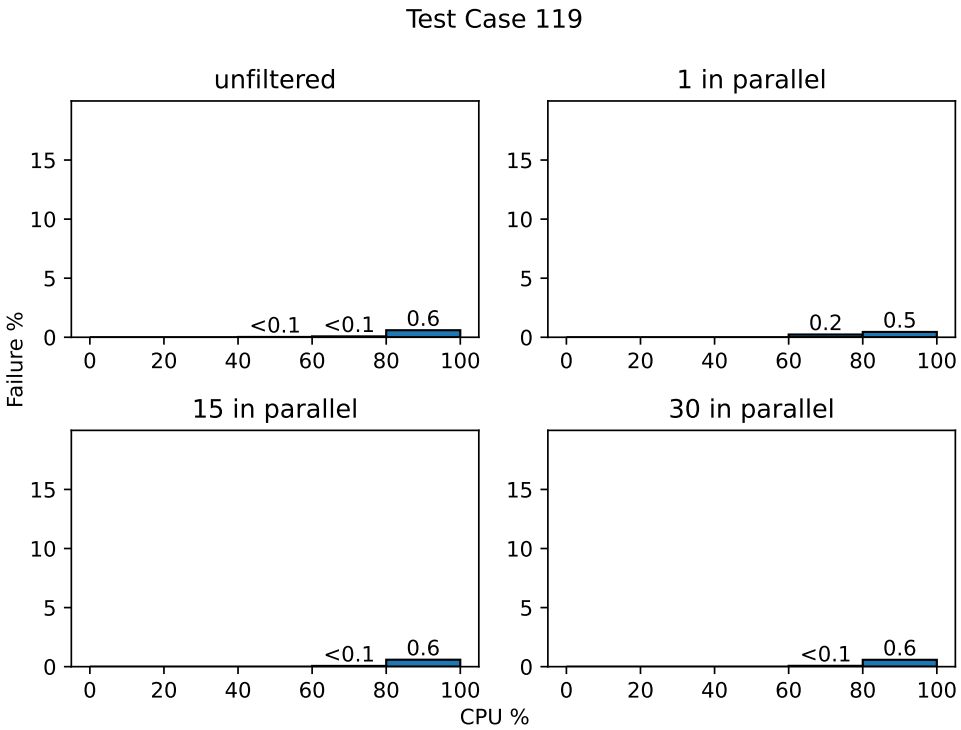


Figure C.10: Test Case 119, failure rate over CPU usage, split by no. parallel executions

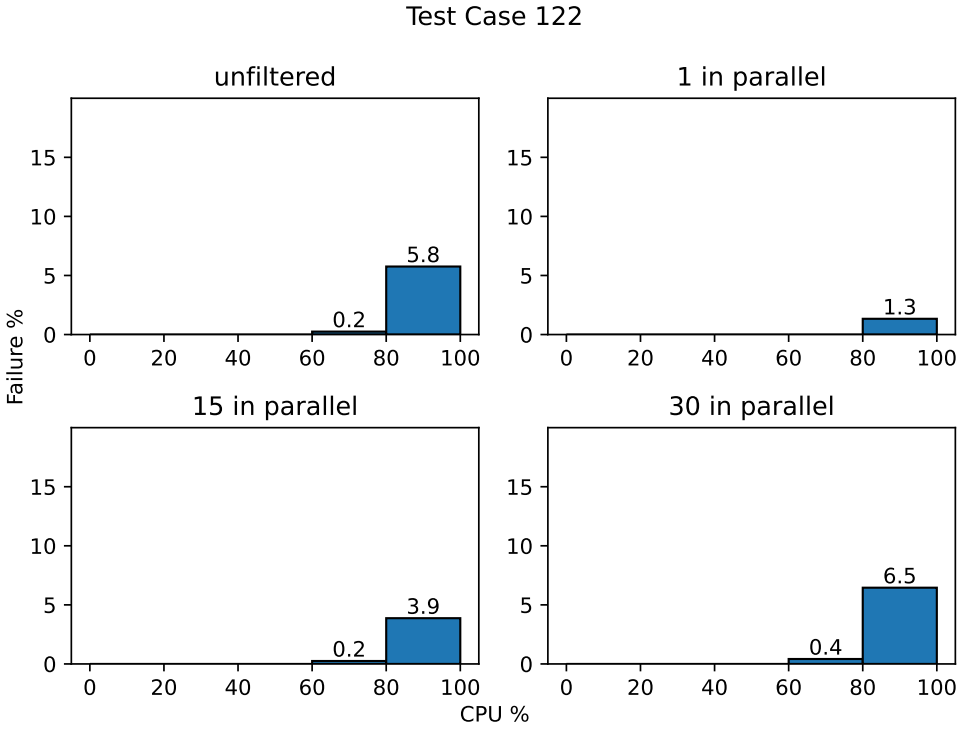


Figure C.11: Test Case 122, failure rate over CPU usage, split by no. parallel executions

D Figures of Code Analysis Output for RQ3

This appendix contains figures detailing the code analysis performed for RQ3. First we present the extracted test smells, and then the code metrics. All figures show their content grouped on the flakiness labels non-flaky, resource-affected flaky test (RAFT) and resource-independent flaky test (RIFT).

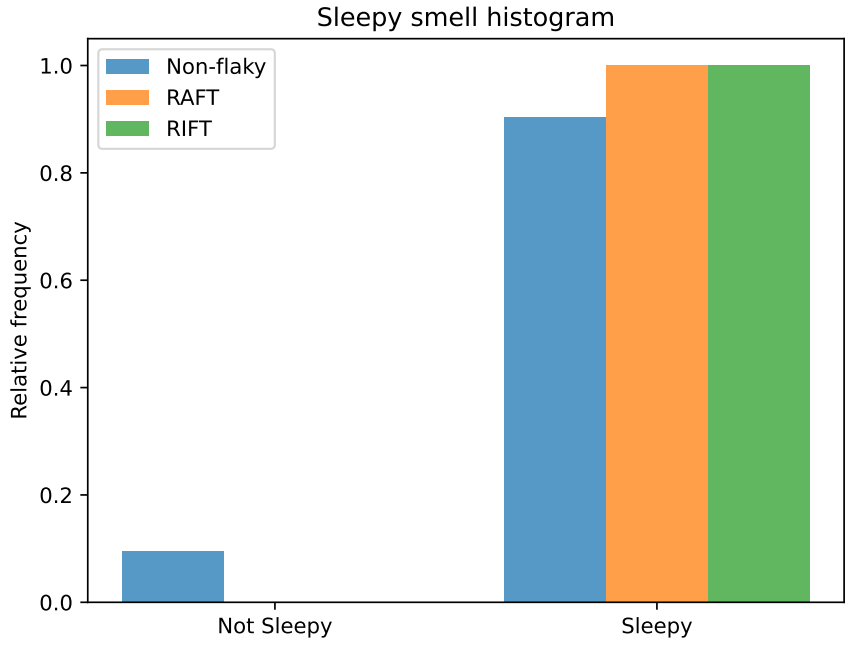


Figure D.1: Histogram of Sleepy Test test smell frequency per flakiness label

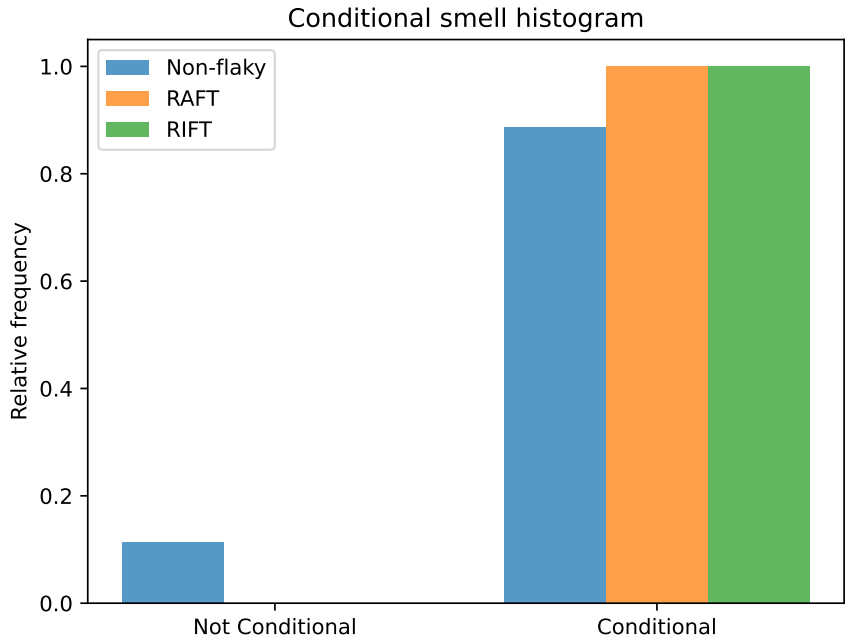


Figure D.2: Histogram of Conditional Test Logic test smell frequency per flakiness label

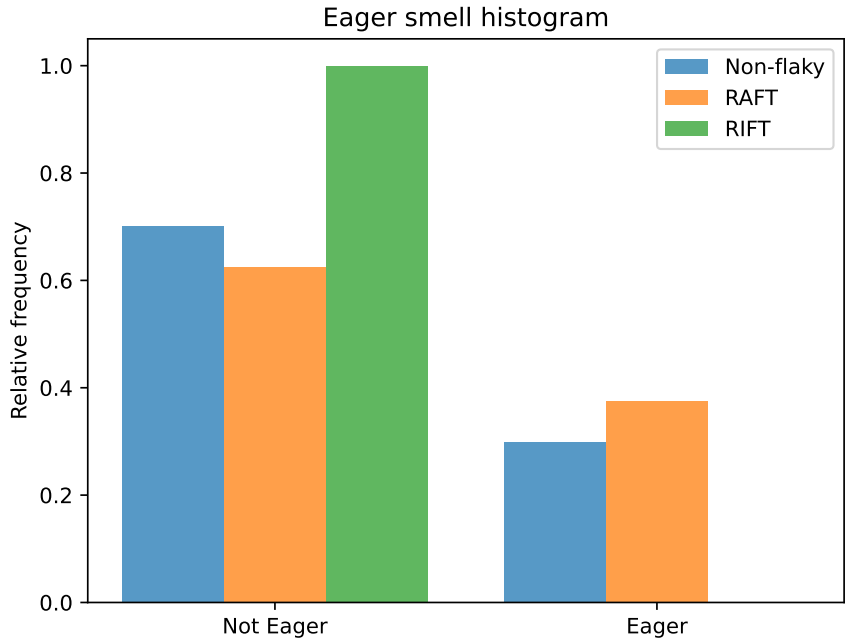


Figure D.3: Histogram of Eager Test test smell frequency per flakiness label

Figure D.4: Histogram of Lazy Test test smell frequency per flakiness label

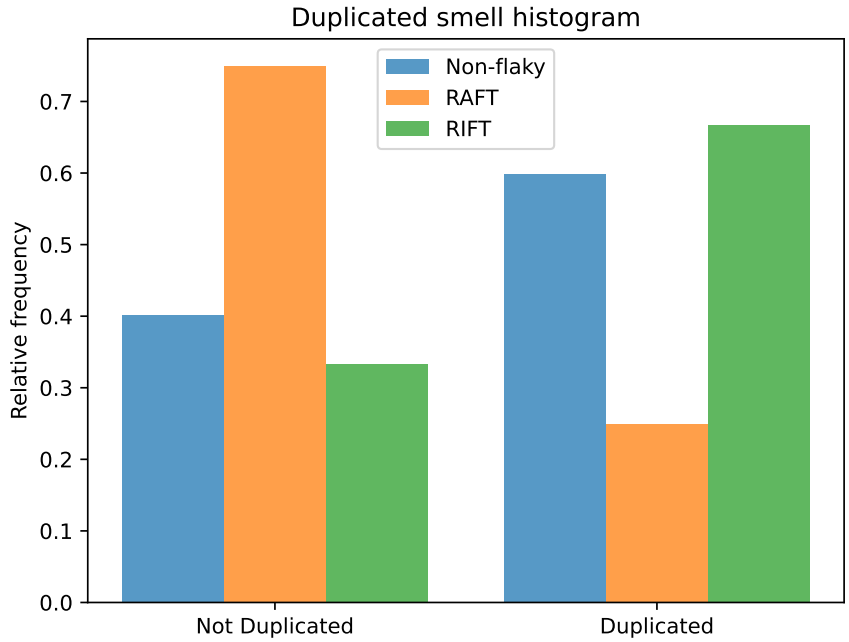


Figure D.5: Histogram of Test Code Duplication test smell frequency per flakiness label

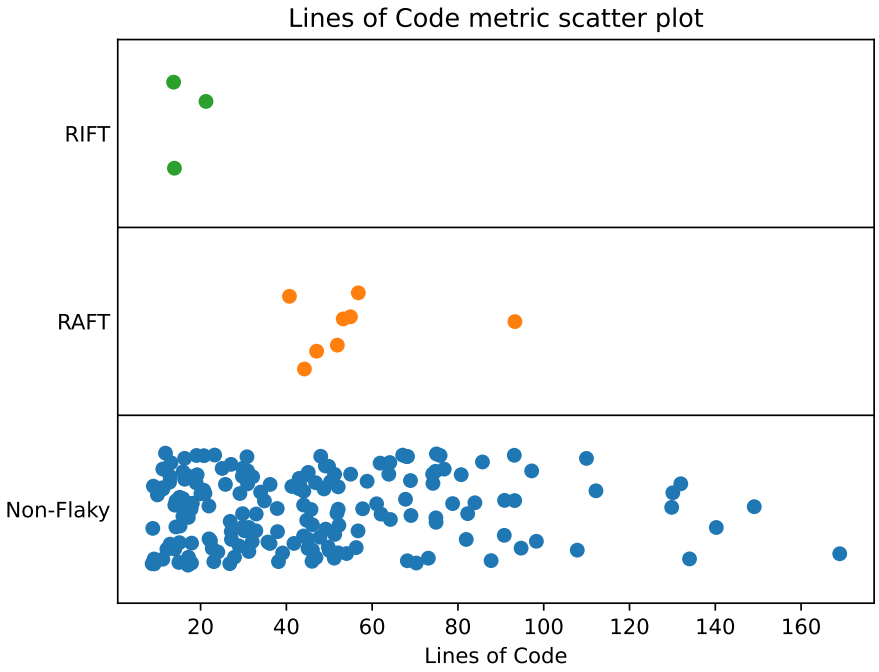


Figure D.6: Scatter plot of Lines of Code per flakiness label

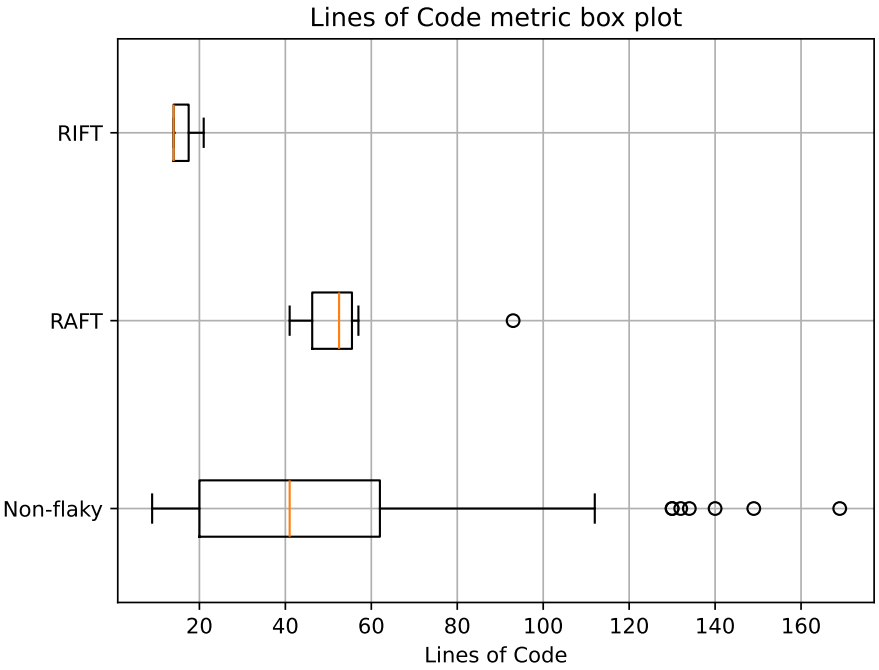


Figure D.7: Box plot of Lines of Code per flakiness label

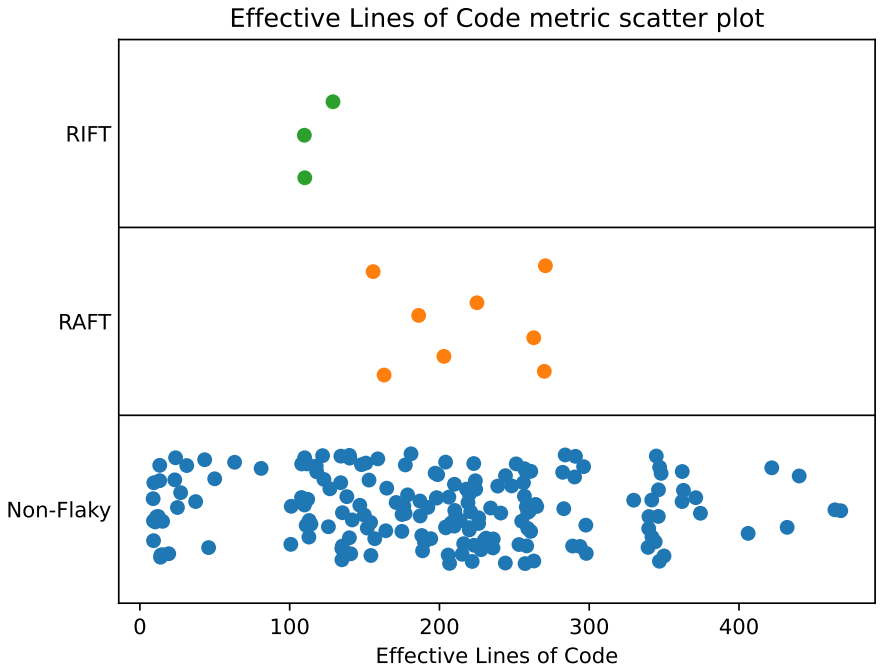


Figure D.8: Scatter plot of Effective Lines of Code per flakiness label

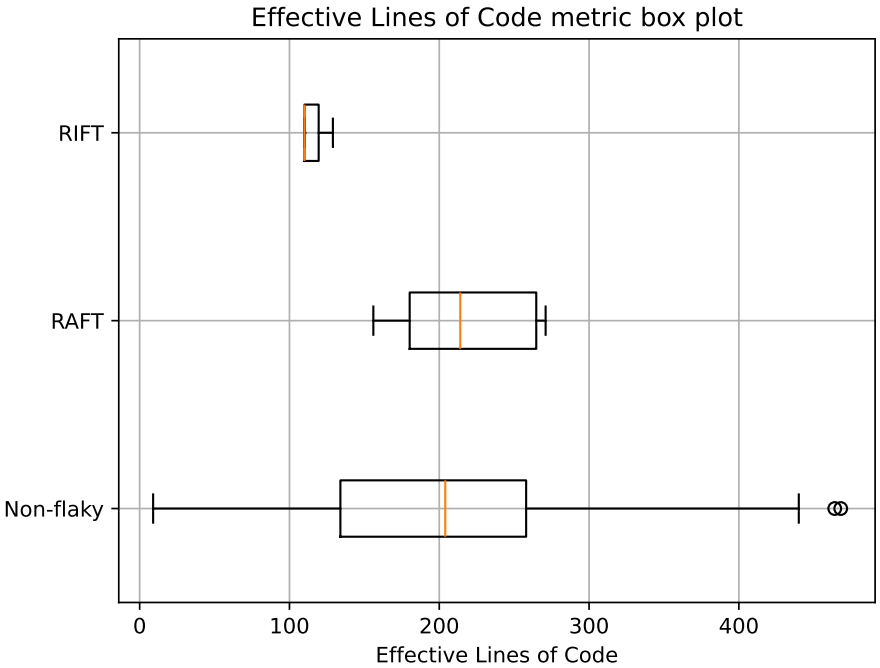


Figure D.9: Box plot of Effective Lines of Code per flakiness label

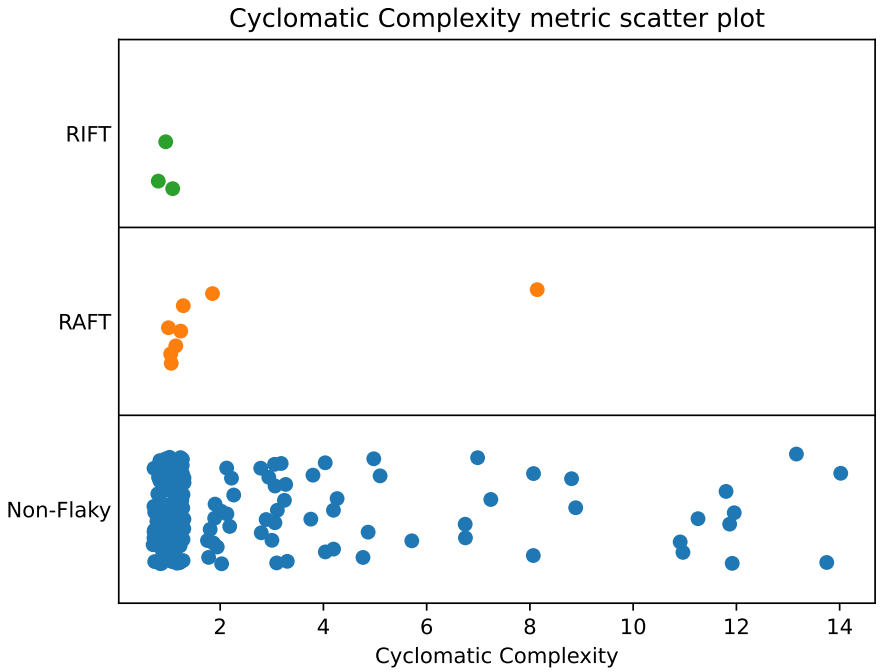


Figure D.10: Scatter plot of Cyclomatic Complexity per flakiness label

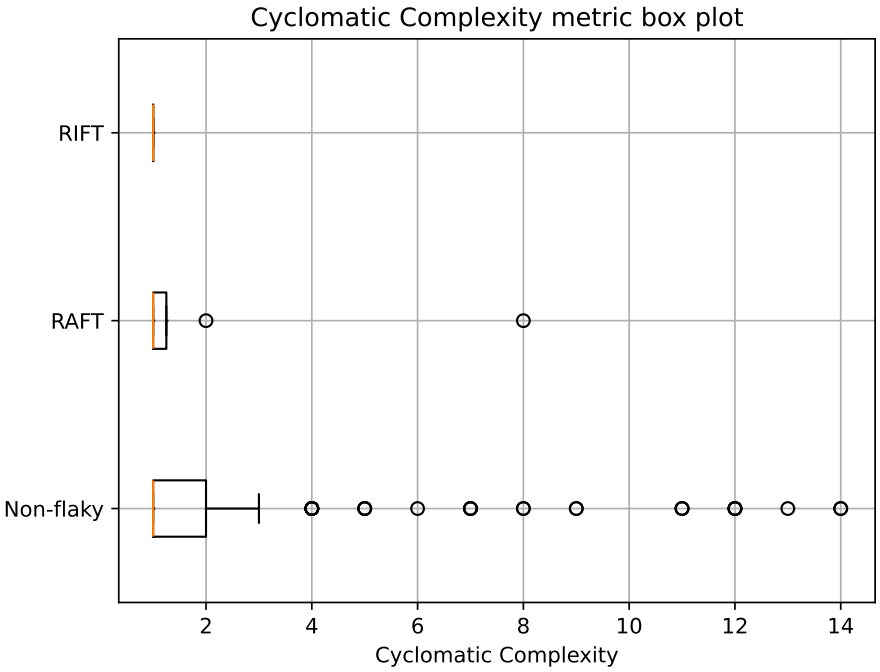


Figure D.11: Box plot of Cyclomatic Complexity per flakiness label

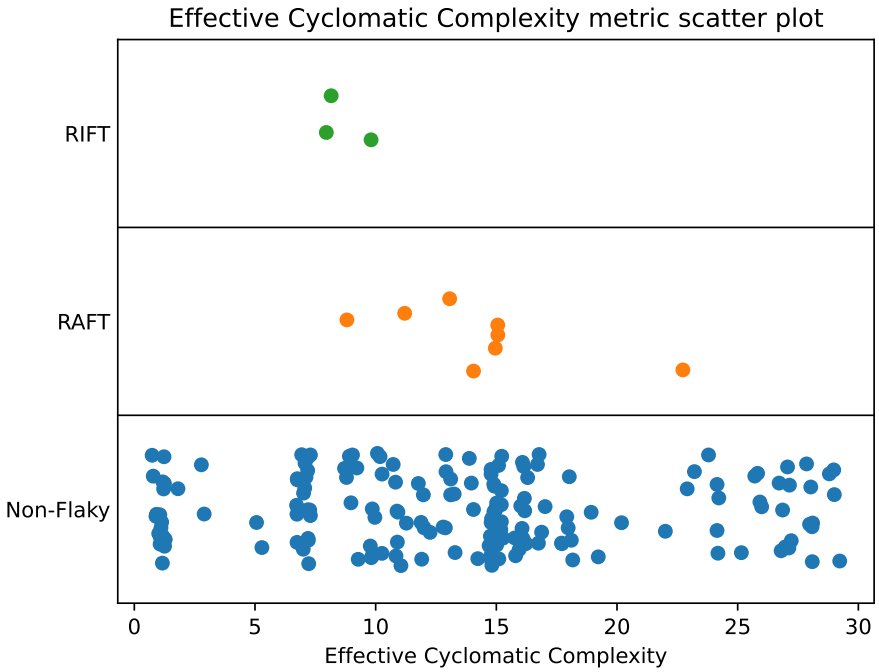


Figure D.12: Scatter plot of Effective Cyclomatic Complexity per flakiness label

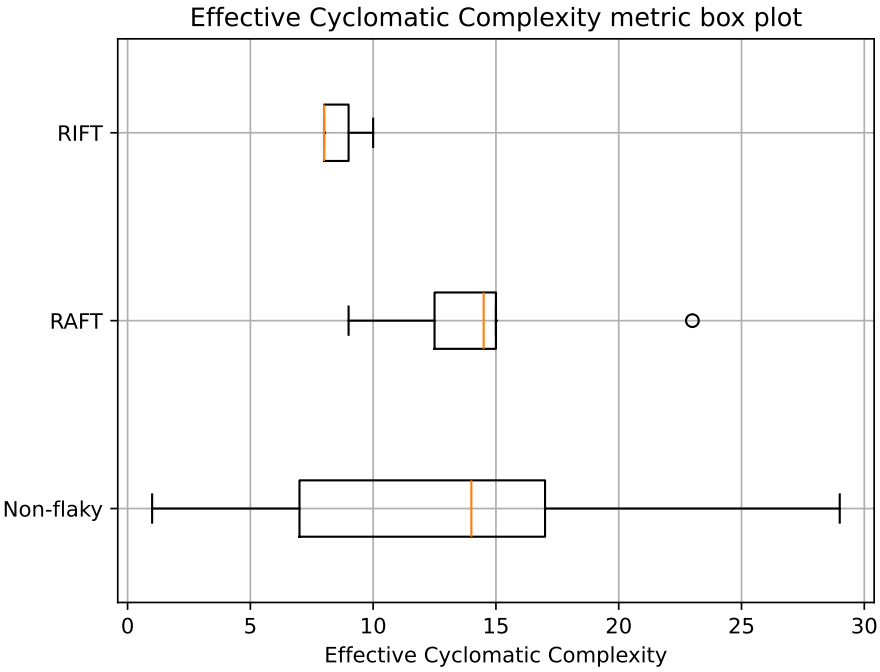


Figure D.13: Box plot of Effective Cyclomatic Complexity per flakiness label

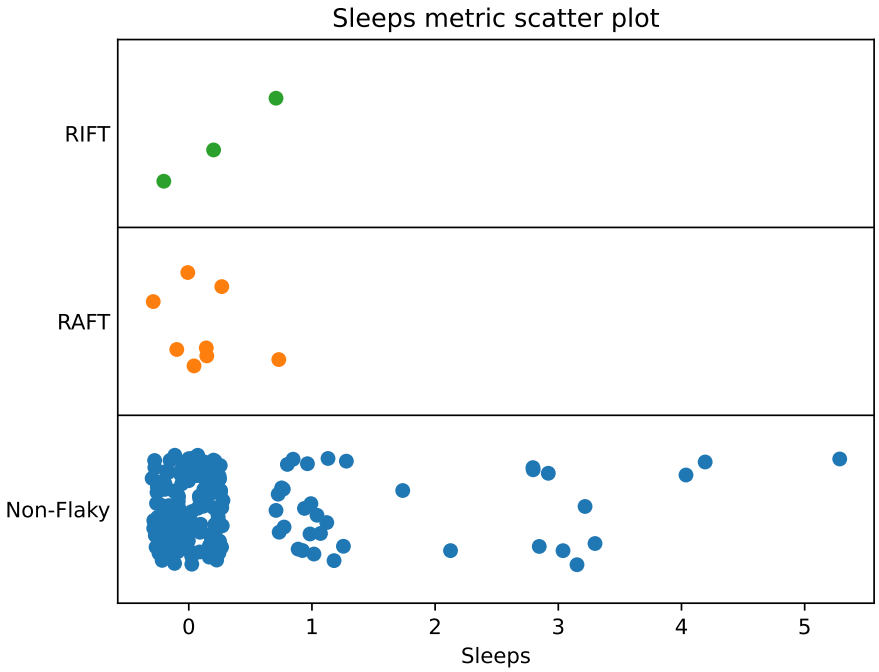


Figure D.14: Scatter plot of Sleeps per flakiness label

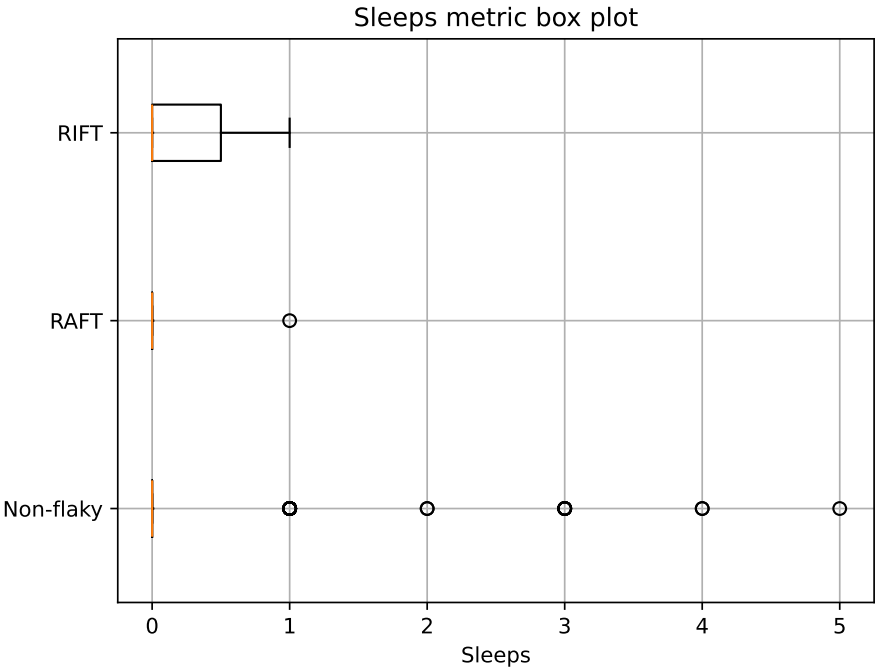


Figure D.15: Box plot of Sleeps per flakiness label

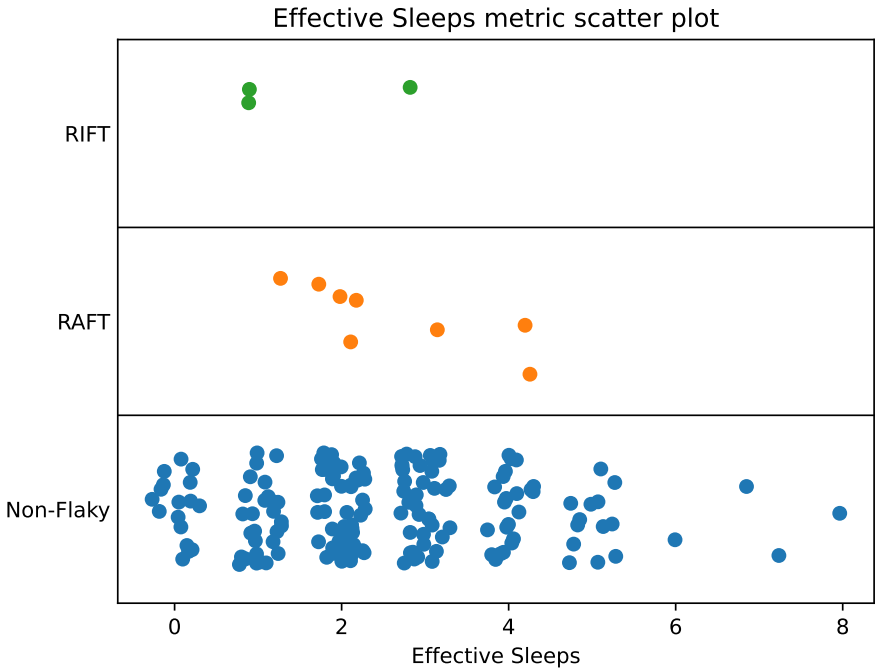


Figure D.16: Scatter plot of Effective Sleeps per flakiness label

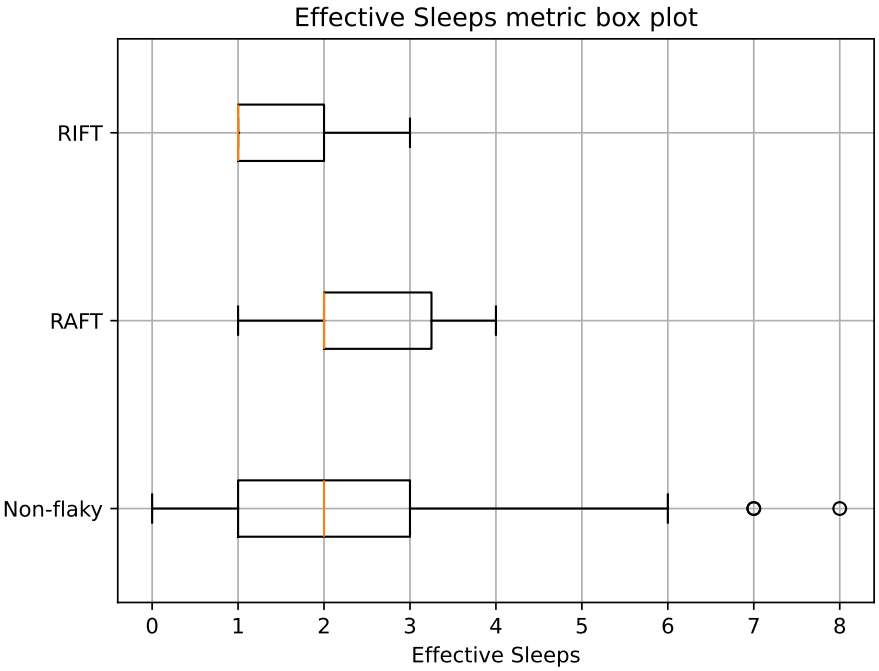


Figure D.17: Box plot of Effective Sleeps per flakiness label

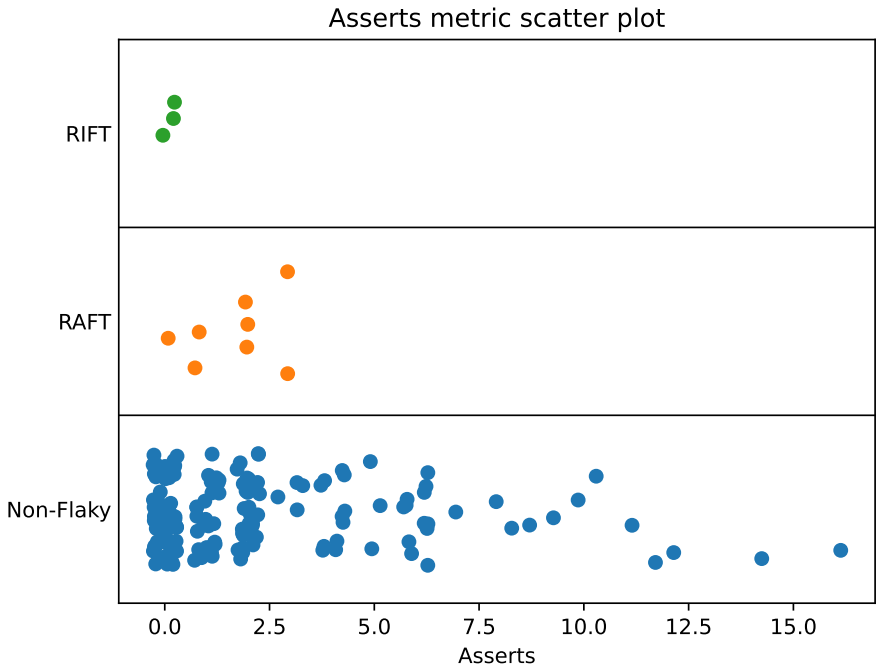


Figure D.18: Scatter plot of Asserts per flakiness label

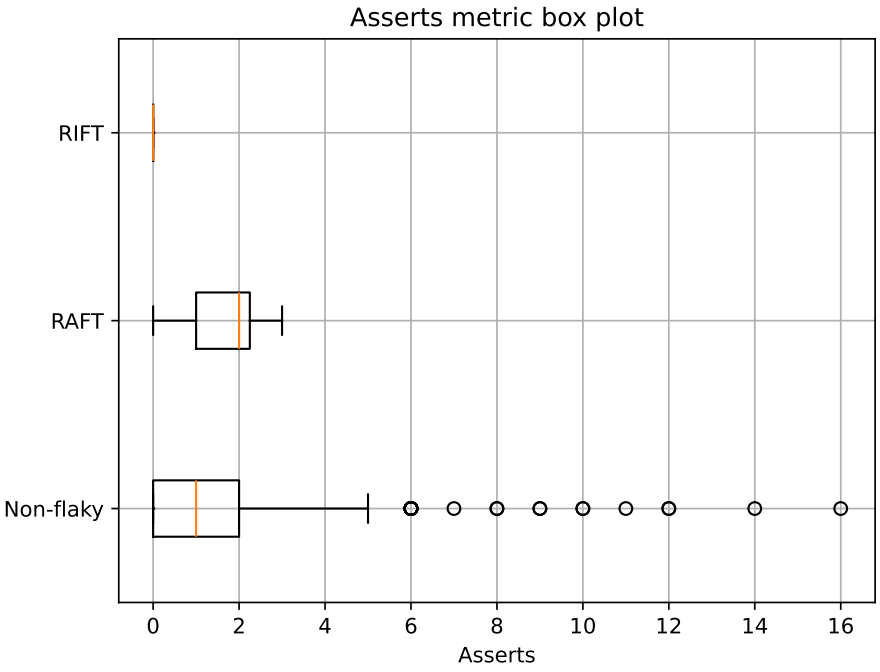


Figure D.19: Box plot of Asserts per flakiness label

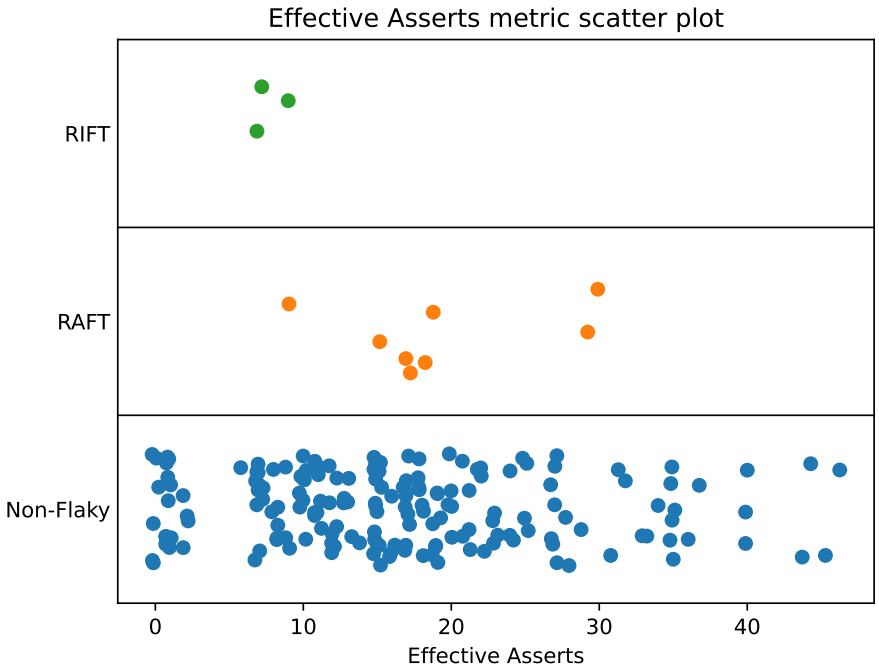


Figure D.20: Scatter plot of Effective Asserts per flakiness label

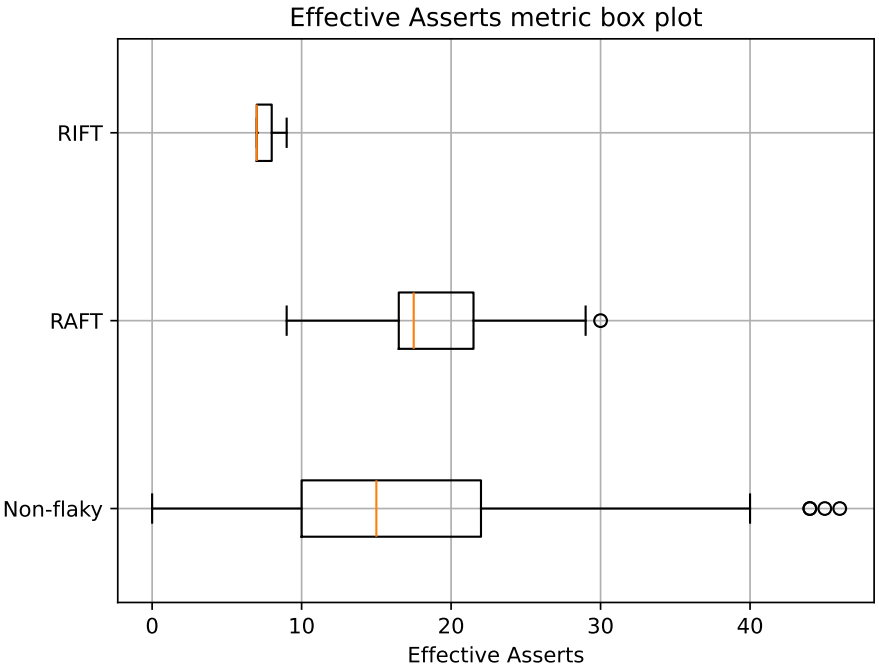


Figure D.21: Box plot of Effective Asserts per flakiness label

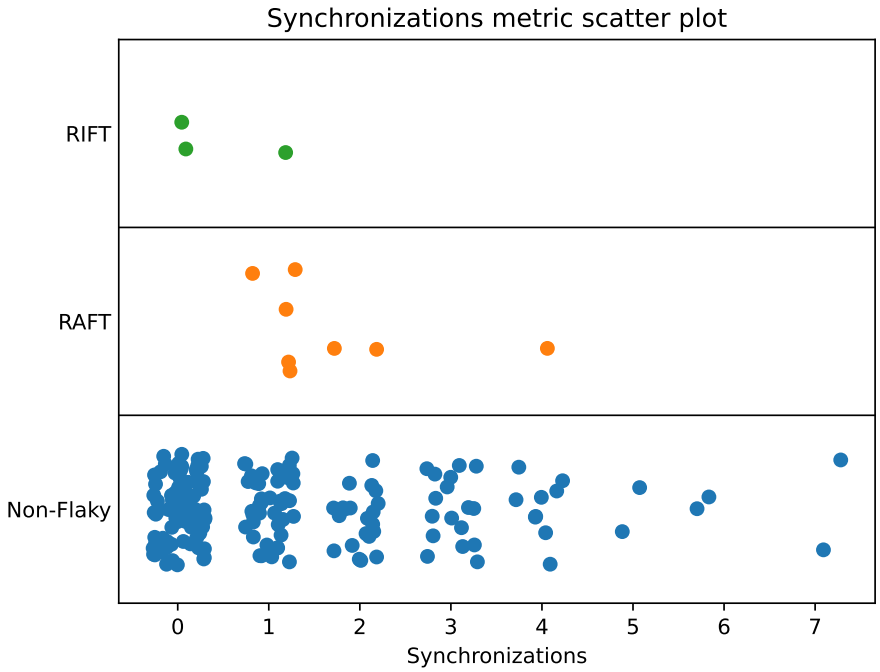


Figure D.22: Scatter plot of Synchronizations per flakiness label

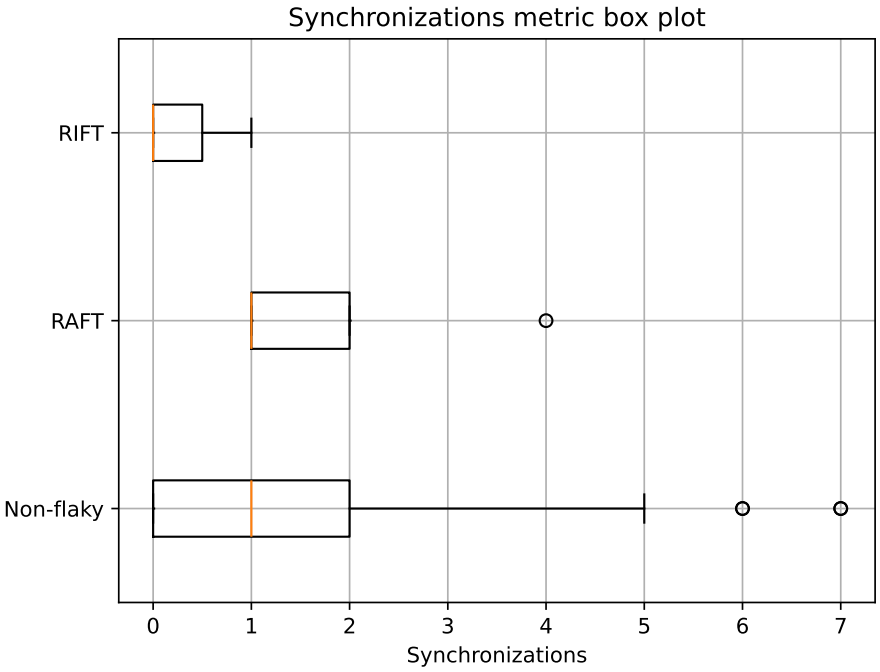


Figure D.23: Box plot of Synchronizations per flakiness label

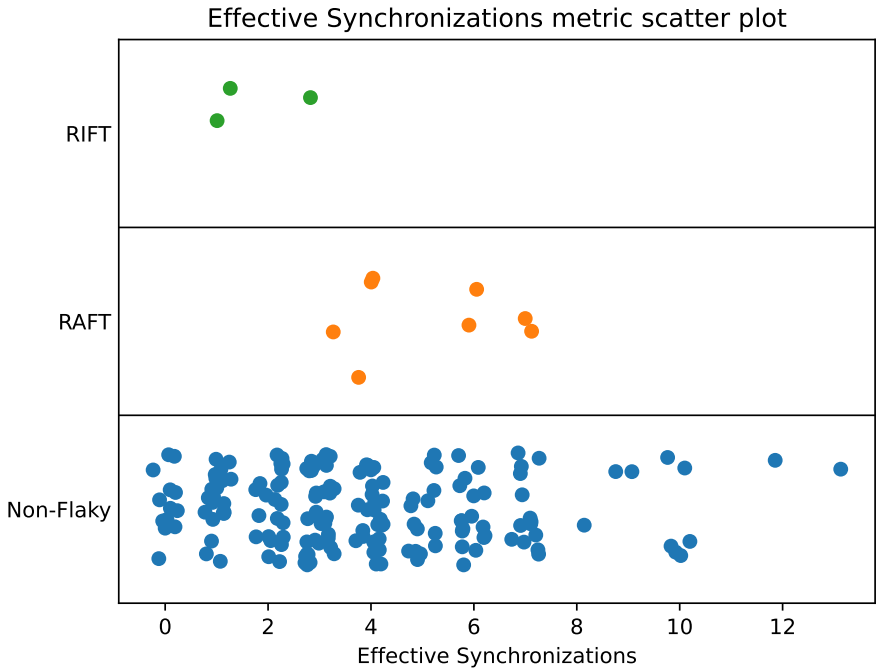


Figure D.24: Scatter plot of Effective Synchronizations per flakiness label

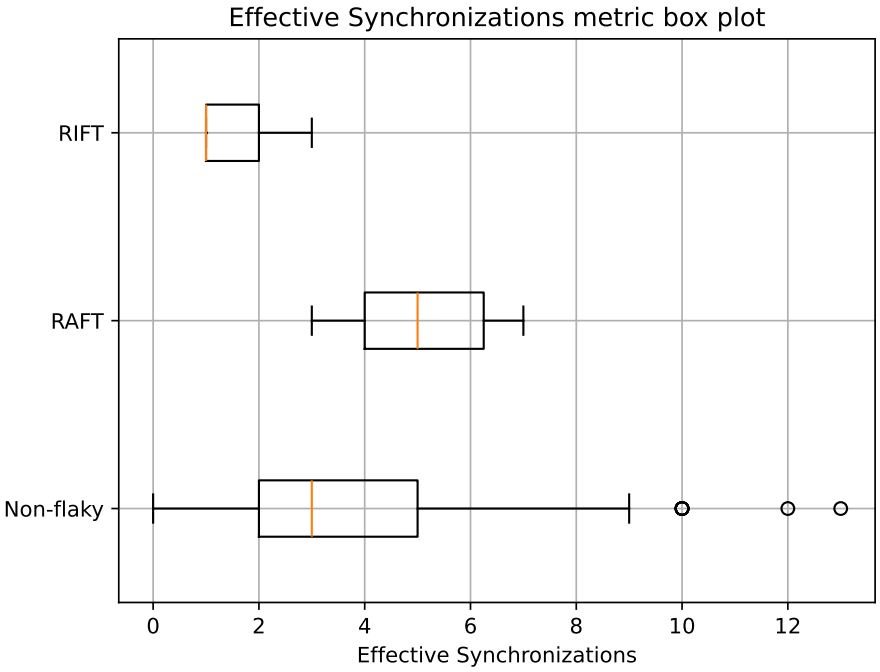


Figure D.25: Box plot of Effective Synchronizations per flakiness label