School of Computation, Information and Technology – Informatics

Technical University of Munich

Master's Thesis in Electrical Engineering and Information Technology

An Empirical Analysis of Flaky Failures in Continuous Integration

Philipp Fink

School of Computation, Information and Technology – Informatics

Technical University of Munich

Master's Thesis in Electrical Engineering and Information Technology

An Empirical Analysis of Flaky Failures in Continuous Integration

Eine empirische Analyse von Flaky Failures in Continuous Integration

Author: Philipp Fink
Submission Date: June 15, 2024
Supervisor: Prof. Dr. Alexander Pretschner
Advisor: Fabian Leinen

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, June 15, 2024 Philipp Fink

Abstract

In today's fast-paced software development environment, teams often use Continuous Integration (CI) to ensure that their code changes are automatically built and tested in a timely manner. A significant bottleneck to the efficiency of developers using CI systems is the presence of flaky tests, which can pass or fail on the same version of code. Flaky tests have been the subject of several existing studies. However, the vast majority of previous work focused on detecting, mitigating, or repairing flaky test cases. We conduct an empirical study in a proprietary software project to analyze how flaky test failures appear in a CI environment where the code and the test infrastructure evolve. Furthermore, we ask whether flaky failures are undetected in the CI and how effective the automatic rerun process is to differ from flaky and non-flaky failures.

To close the research gap, we collect historical data from our industrial partner's CI. After two months with $\sim$ 19 million test executions, we conducted an empirical analysis of the flaky failures, which the CI detect after rerun specific tests on failure. These failures exhibited various features, including test results and failure symptoms, which could include error messages and stack traces. Our findings revealed that 0.10% of test executions were flaky, with three test cases collectively responsible for approximately 80% of all flaky failures. Subsequently, we interviewed developers responsible for the eight test cases exhibiting the most flaky failures. The objective was to ascertain the underlying causes of these failures and the nature of the associated tests. The majority of developers attributed the flakiness of these tests to the CI environment, with the infrastructure identified as the primary origin of the issue. In addition, the developers indicated that seven of the eight tests were tests with a larger scope (e.g., system tests, integration tests, or component tests). To distinguish between flaky and regression failures, the CI system uses reruns. However, the efficacy of this method remains uncertain. We aim to evaluate the effectiveness of reruns in identifying flaky failures. Therefore, we compare failure symptoms from failures that occur after rerunning with failure symptoms from flaky failures. We find that 4.5% of the failures are wrongly identified as regression failures. In addition, we compare the fail rate of different pipelines using rerunning with the fail rate without rerunning. Depending on the pipeline, we find that the fail rate would increase by up to 23% without rerunning.

Our study provides first insights to analyze flaky failures more than flaky test cases, as they reduce the efficiency of developers and the CI. Furthermore, other researchers can use our findings to compare with other proprietary and open-source projects with different CI settings to generalize the results.

Contents

1 Introduction

In software development, large-scale projects often involve a multitude of developers working concurrently on different features and fixes. In such an environment, developers find manually integrating these changes into a shared repository impractical. It takes a lot of time and money for developers to coordinate with each other. Furthermore, no matter how small, every change can cause a regression, which is the reintroduction of previously resolved issues or the introduction of a new bug [28, 36, 50]. To address these problems, the software engineering community has embraced the concept of CI as a cornerstone. CI is a development practice that allows developers to automatically integrate code into a shared repository [11]. An automated process then verifies each commit, allowing teams to detect problems early. These tests ensure that the new changes do not break or degrade the system's existing functionality. As long as the code under change remains unaltered, regression tests should produce consistent outcomes—pass or fail. However, the assumption of determinism is only occasionally accurate. Tests that produce inconsistent test results on different occasions, where all the environmental factors the tester can control are the same on each execution, are called flaky tests [24]. CI hinges on the stability of automated tests, but flaky tests can severely disrupt this process by introducing uncertainty and inefficiency. Addressing flaky tests is, therefore, essential to maintaining the integrity and effectiveness of the CI pipeline [23, 42].

Since the initial investigations into flaky tests [38], there has been a substantial commitment by both researchers and industry professionals to better understand [15, 38] and counteract the adverse effects of flaky tests. Efforts have included creating tools for automatic detection [8, 21, 22], pinpointing underlying root causes [1, 33, 40], and formulating strategies to fix flaky test cases [13, 51]. Researchers have devoted a large portion of the research to date to examining flaky test instances. They typically identified these instances by repeatedly rerunning test cases in controlled settings [21, 31] or by analyzing commits that repair flakiness [25, 38, 43, 46, 52]. Although the studies mentioned above offer essential information about flaky tests, they consider each case equally problematic. However, the failures that occur through flaky test cases are the main reason that hinders the CI process [23, 42].

To the best of our knowledge, we are the first to empirically study flaky test failures in the context of a CI environment. Therefore, we assembled a dataset from our automotive industry partner, BMW. The steps included collecting the results of tests, jobs, and pipeline executions. Our dataset contains 27,245 test cases, 714, which show flaky behavior. All test cases have been executed 19,594,834 times in the CI environment, resulting in 304,575 failure where at least 22,006 are flaky identified by rerunning, and 19,292,215 passed ones. In particular, we address the following research questions (RQs):

RQ1. *How does flakiness occur in a CI system?*

We empirically analyze the collected data to study the characteristics of flaky test failures in CI setting. To better understand test cases with a high frequency of flaky failures, we consulted with experts about the test type, root cause, and sources of flakiness. Therefore,

we clustered identical failure symptoms, including error messages and stack traces. For a better structure, RQ1 is divided into four smaller questions:

RQ1.1. *Which share of flaky test cases is responsible for a certain share of flaky test failures?*

RQ1.2. *What are the root causes of flaky failures?*

RQ1.3. *Where do the flaky failures originate from?*

RQ1.4. *What types of tests are responsible for flaky failures?*

In the CI environment, 0.10% of the test executions exhibit flaky failures. These flaky failures are distributed across 714 unique test cases, representing a ratio of 2.6% of all tests. The flaky failures are very unevenly distributed across flaky tests. Three test cases are responsible for approximately 80% of all flaky failures. Most flaky failures originate from the infrastructure and are integration tests where the root cause is mainly due to the CI.

RQ2. *To what extent do flaky failures survive rerunning?*

One cause of failed pipelines are test failures, some of which fail even after several reruns. These test failures are potentially flaky, yet this assumption can only be validated through additional executions. However, we compared existing flaky failures with failures that fail for all reruns to estimate the proportion of them that *survive* the rerun process without showing flaky behavior. Our findings indicate that 4.5% of the failed test executions, which were rerun, are potentially flaky.

RQ3. *How effective is rerunning?*

The technique of rerunning tests on failure is an established method for detecting flaky failures. To gain a deeper understanding of the effectiveness of rerunning, we calculated the flake rate of pipelines with and without the implementation of rerunning on failure. We found that without rerunning, the fail rate of pipelines increased in the range from 3.7% to 23%.

The rest of this thesis is organized as follows: Next, in Chapter 2, we give some background information on CI, flaky tests, and flaky test failures. In Chapter 3, we review related work. Our study setup, including the CI environment at BMW, data collection, abstraction of failure symptoms, and how we organized the interviews, is described in Chapter 4. In Chapter 5, we present our findings and answer our research questions. Afterward, we summarize the results in Chapter 6 and provide recommendations for action to our industry partner. We discuss the limitations of our approach in Chapter 7 and summarize our findings and potential future work in Chapter 8.

2 Background

This chapter provides essential background information for understanding the context of this thesis. We introduce the concept of CI, explain flaky test failures, the root causes of flakiness defined in the literature, and the impact of flaky tests.

2.1 Continuous Integration

Since its introduction by Booch in 1990 [9] and subsequent adoption as a key practice of Extreme Programming in 1999 [7], CI has become a cornerstone in software engineering. CI encourages frequent integration and merging of work, often multiple times a day, and leverages automation for building, testing, and deployment processes. The goals of CI include shortening development cycles, enhancing the reliability of software releases, increasing customer satisfaction, simplifying bug detection, and boosting developer productivity and collaboration [11, 27, 44, 45, 49].

The widespread adoption of CI across various sectors, from the open-source community to large-scale enterprises, demonstrates its efficacy. By 2016, CI was utilized by 70% of the top 500 open-source projects on GitHub [27]. Travis CI, particularly favored in open-source development, has been implemented in over 300,000 [1] projects. GitHub Actions, introduced in 2019, quickly became the leading CI service on GitHub, with over 400,000 repositories adopting it within 18 months of its release [19]. Meanwhile, Google reports 150 million test executions daily, with over 13,000 teams merging their work into a single repository daily [39].

CI typically consists of four main components besides the developers: (1) a version control repository, (2) a CI server, (3) an automated build tool, and (4) a feedback mechanism [48]. Developers are expected to follow several practices, such as creating automated tests for the code they write, committing code frequently, and prioritizing fixing broken builds before moving on to other tasks. The version control repository is a central repository for all code and asset changes developers make. It maintains historical versions of the project and various active development branches. The role of the CI server is to monitor or be notified of every change committed to the repository. It fetches the source files and looks for a build configuration file. While there may be multiple files, the build configuration should contain, or at least point to, instructions on how the CI server should compile, build, configure, and run the appropriate tests. Several functional tests are performed, ranging from small unit tests, over integration tests where different components are tested together to more extensive system tests that check the overall system requirement [28]. However, all types check if the code meets the required functionality and behavior. Once the tests have been run, the CI server uses the feedback mechanism to communicate the build results. This system is responsible for

[1] Metrics from `https://www.travis-ci.com/about-us/`, visited on 2024-04-17

informing the developers of the outcome of the build so that they can quickly resolve any issues, such as failed tests or other problems that may have arisen. The developer then commits the necessary fixes to the repository, continuing the CI cycle.

2.2 Flaky Tests and Failures

In software testing, the term flaky test is subject to various definitions as found in existing literature [5]. Despite the differences, most of these definitions converge on a key characteristic of flaky tests: they exhibit inconsistent outcomes over repeated executions, even when specific factors are constant.

For this thesis, we will adopt the definition provided by Harman et al., which articulates that a *"flaky test is one for which a failing execution and a passing execution are observed on two different occasions yet for both executions, all environmental factors that the tester seeks to control remain identical"* [24]. Based on this definition, we define flaky failures as test failures that pass on any rerun (with $lim_{n\to\infty}$) without changing the environmental factors the tester seeks to control. In this thesis, we will investigate flaky failures, but their existence also means that the corresponding test is flaky.

2.2.1 Impact of Flaky Failures

Ignoring flaky failures could be a problem as they may hide real bugs. Luo et al. [38] found that 24% of flaky test were fixed by also changing the Code under Test (CUT).

In the context of CI, the impact of flaky failures can be significant. The reduction in reliability can affect the overall efficiency of the development process. Since ignoring flaky test failures may harm software stability [44] and may hide real bugs [38], developers often find themselves needing to manually investigate these failures, which can disrupt their workflow and slow down progress. However, one of the challenges in investigating flaky failures is that they are difficult to reproduce due to their non-deterministic nature [32, 33]. This unpredictability can lead to a considerable amount of time spent trying to pinpoint the cause of the failure. Furthermore, flaky failures can erode trust in the testing suite, leading developers to ignore test results over time [15].

2.2.2 Root Causes

Researchers have attempted to ascertain the underlying causes of the inconsistent test results for some time. Luo et al. [38] performed one of the earliest empirical studies of test flakiness. They categorized the cause of the flaky tests repaired by developers in 201 commits across 51 open-source projects using the following ten categories:

1. Async. Wait. A test that makes an asynchronous call can experience flaky failures if it does not correctly wait for the call to complete.

2. Concurrency. A test that creates multiple threads can encounter failures if those threads interact unexpectedly, such as creating a race condition.

3. Floating Point. Tests that involve floating-point calculations may become flaky when

they yield unpredictable results.
4. Input/Output (I/O). Tests interacting with the file system can be flaky due to external factors, such as limited storage space.
5. Network. Tests may fail flaky when the network is down or congested.
6. Order Dependency. Tests that rely on a specific execution order or shared resources, which other tests may alter.
7. Randomness. Tests that use random number generators without fixed seeds.
8. Resource Leak. Tests can also become flaky if they fail to release resources, like database connections, affecting subsequent tests that need those resources.
9. Time. Tests are susceptible to flakiness due to differences in how time is measured or represented across various libraries and platforms.
10. Unordered Collection. Tests that assume a consistent iteration order in unordered collections, such as sets, may fail intermittently when that assumption is proven incorrect.

Later, Eck et al. [15] surveyed several professionals to classify previously fixed flaky tests following the taxonomy developed by Luo et al. However, the possibility of adding new categories was retained in case of necessity. *Too Restrictive Range*, *Test Case Timeout*, *Test Suite Timeout*, and *Platform Dependency* were added. While studying flakiness in Python tests, Gruber et al. [21] identified *Infrastructure* as another root cause.

2.2.3 Undetected Flaky Failures

A prevalent method for distinguishing between regression and flaky test failures involves rerunning tests after an initial failure. This process, employed by companies like Google [39] and Microsoft [33], involves rerunning failed tests to determine if they subsequently pass, suggesting a flaky failure rather than a regression. The chosen number of reruns often lacks a standardized basis, with some opting for a convenient figure like ten. Lam et al. [34] observed that using five or fewer reruns can still accurately identify flaky tests in various Java projects with an accuracy of at least 82%. In another study, Leinen et al. [35] discovered that without increasing the number of reruns, 4.8% of all pipelines would have erroneously failed due to undetected flaky tests.

This thesis aims to explore test failures that persist through all reruns to gain insight into the frequency of potentially flaky failures that have not been identified. These undetected cases are referred to as *survived* failures within this research.

3 Related Work

This chapter provides an overview of existing work related to this thesis. First, it presents existing studies about flaky tests. Second, it reviews existing work about flakiness in the context of CI. Finally, it presents the research gap that this thesis aims to fill.

3.1 Studying Flaky Tests

In recent years, test flakiness has captured increasing attention within research circles. The earliest study on this subject was conducted by Luo et al. [38], who analyzed 201 commits from 51 open-source Java projects. Through manual inspection, they categorized the root causes of flakiness into ten distinct categories and showed that async wait, concurrency, and test order dependency are the main categories of flakiness. Lam et al. [33] later analyzed 55 Java projects to comprehend the introduction of flakiness in software systems. They discovered that 75% of the 245 detected flaky tests were already flaky upon their addition to the test suite. This justified the necessity to run detectors on newly introduced tests. However, the scope of research on flakiness extends beyond the Java programming language. For instance, Gruber et al. [21] empirically analyzed flaky tests within the Python programming language. By executing each test suite 200 times in both its original and a randomized order, they could identify tests that exhibited flaky behavior. In total 0.86% of 8,080,349 test cases across 1,006 projects were flaky. By running the 200 runs in 10 iterations of 20 runs each, this work introduced an additional root cause of flakiness termed Infrastructure. The 10 iterations were distributed across different machines. If the verdict between iterations changed, the test would be considered flaky. Similarly, Hashemi et al. [25] conducted an empirical study on flaky tests in JavaScript by analyzing 452 commits from large, high-scoring JavaScript projects on GitHub. The study highlighted concurrency and asynchronous wait as leading causes of flakiness in JavaScript. Research has focused on different programming languages and flakiness factors specific to certain applications or domains. Thorve et al. [52] conducted a study specifically on Android projects, examining historical commits and identifying the exact root causes of flakiness as [38] Dutta et al. [14] examined 75 fixed flaky tests from commit and bug-report data to categorize causes and fixes of test flakiness in machine learning projects written in Python, noting that 60% of the root causes could be classified under the category of randomness as defined by the taxonomy from [38]. In another study, Eck et al. interviewed 21 software developers from Mozilla to classify 200 flaky tests they had previously fixed. They uncovered four new emergent categories: test case timeout, test suite timeout, platform dependency, and overly restrictive range. The interviews also revealed the issue's prevalence, with 20% of respondents encountering test flakiness monthly and 15% dealing with it daily. Despite the diversity of these studies, they all share a common methodology: investigating flaky tests by utilizing historical commits, bug reports, fixed flaky tests, pull requests, or execution traces to identify and understand the nature of test flakiness.

The academic community has extensively explored identifying flaky tests, yet the differentiation between flaky and actual test failures has not been equally addressed. Only a few studies have used information from flaky test failures to decide whether a failure should be disregarded or investigated further. Alshammari et al. [2] conducted a study utilizing a dataset of 498 flaky tests from 22 open-source Java projects, along with 230,439 failure messages, to study the effectiveness of failure de-duplication techniques. They implemented text-based matching and simple machine learning classifiers after de-duplication. They found that the success of these methods greatly depends on the specific test and the type of failure involved.

In another study, An et al. applied abstracted data from error messages and stack traces to classify test failures as flaky within the SAP HANA database [4]. They refined failure symptoms by masking numbers and purifying stack traces, such as omitting line numbers and test entry points. Their methodology achieved a high level of accuracy, with a reported precision of 96% and recall of 76% in identifying flaky failures.

3.2 Flakiness in CI

Few studies have specifically tackled the challenge of flaky tests in CI environments, yet their contributions are valuable for understanding the issue. Micco reported that within Google's CI pipeline, 1.5% of all test executions are flaky, with about 16% of all tests exhibiting flakiness [39]. Lam et al. developed a comprehensive framework to identify flaky tests and diagnose their root causes by analyzing the differences between logs from passing and failing runs [33]. In their approach, they reran tests flagged as flaky by the CI 100 times in a controlled local setting and found that 86% of these tests were flaky only within the CI context. They also noted that while flaky tests are relatively few, they contribute to a significant number of build failures in Microsoft's CI system. Leinen et al. utilized historical CI data, including test logs, version control commits, issue tickets, and recorded work hours, in an industrial case study to evaluate the costs related to flaky tests [35]. They concluded that the costs of rerunning flaky tests are relatively low compared to the resources required for their analysis and resolution. Chen et al. investigated test executions and failures in CI by examining a dataset tracking test outcomes and code coverage across 12 open-source Java projects [10]. They found that 56% of the test failures were flaky. Before presenting their method for detecting flakiness through the abstraction of failure symptoms, An et al. revealed that 87% of the test suites they studied with failed tests included flaky test failures [4]. Additionally, they observed that the flaky failure symptoms are distinct from those of failing non-flaky tests.

3.3 Research Gap

Previous studies have primarily focused on identifying flaky tests through various methods. These methods include the analysis of historical commits [38], bug reports [53], previously fixed flaky tests [15], and pull requests [32]. Additionally, the practice of

repeatedly rerunning test cases in a laboratory environment is commonly employed to identify tests that exhibit flakiness [21, 31].

This study addresses two assumptions commonly made in such studies: (1) Every flaky test case is considered equally problematic, regardless of the frequency with which the test fails flakily within a specific time frame. (2) The data is captured at a single moment and primarily examines the technical dimensions, often overlooking the procedural elements of the CI. Furthermore, only a few studies specifically address the issue of flaky tests within CI systems [4, 10, 33, 35].

Our study represents a pioneering effort in leveraging historical data from a real-world industrial CI system. By examining instances of flaky failures, we aim to gain an empirical understanding of flakiness within a CI context.

4 Study Setup

To investigate flaky failures, we collected data from an industrial CI environment that exhibits flakiness. Therefore, the first section of the study examines the CI environment at BMW, focusing on its operations and methods for detecting and managing flaky tests and failures. Afterward, the study's data collection and storage process will be elaborated upon to analyze the data later. Finally, we will discuss our method to abstract the failure symptoms gathered for more in-depth analysis. Additionally, we will describe the interview process with experts to gain a deeper understanding of flaky failures.

4.1 CI Environment

In software development for driver assistance and automated driving, BMW has adopted a strategy for managing its extensive codebase by utilizing a single repository for the complete codebase, also known as monorepo. This monorepo primarily focuses on the code for Electronic Control Units (ECUs), which is integral to the operation of modern vehicles, controlling everything from engine timing to driving dynamics. To organize the projects within this monorepo, BMW employs Git submodules [18]. This approach allows for a modular structure where different repositories are nested as subdirectories. To manage code visibility, the repository is split into 38 submodules. However, testing is conducted centrally and synchronously, so this detail can be neglected for this thesis.

The codebase contains $\sim$ 70 million Lines of Code (LOC), which are managed by a team of $\sim$ 6,500 developers. In terms of programming languages, BMW's developers primarily use C++, Python, and C. To ensure the quality and consistency of the code, BMW has implemented a cloud-based CI system using Zuul [2] in combination with Github [3]. This system is a critical component of the software development lifecycle, automatically testing and integrating new code contributions.

4.1.1 CI Setup

Zuul, an open-source project gating system, works closely with Github and thus aims to minimize the change of merging broken code. Zuul introduces the concept of jobs, which are steps defining the actions taken during the CI process. These steps include compiling the code, running tests, and deploying applications. Another core concept in Zuul are pipelines. Pipelines in Zuul are a series of jobs executed in parallel for multiple changes and triggered at specific times throughout the lifecycle of a pull request. These pipelines conduct automated quality assurance at different stages of the code development process. The most neuralgic points are pre-merge, merge queue, and post-merge. Figure 4.1 shows

[2] `https://zuul-ci.org/`, visited on 2024-04-13
[3] `https://github.com/`, visited on 2024-04-13

the complete process, from creating a branch to merging changes into the main branch and the interaction between Zuul and Github.

The pre-merge pipeline triggers before merging a pull request into the main codebase. This pipeline's primary goal is to provide developers with fast feedback, ensuring developer velocity. Consequently, only small and fast tests are included in this pipeline, allowing for quick identification and resolution of issues without significant delays. If the pull request passes the pre-merge checks and receives a positive code review from a code owner, it goes into the merge queue. Zuul speculatively merges multiple changes in this stage using a First In, First Out strategy. Testing speculative merged changes ensures that the combination of concurrent changes does not break existing functionality.
In contrast, an independent pipeline, the post-merge pipeline, runs after merging changes into the main codebase. An independent pipeline in Zuul processes each job without depending on other jobs within the same pipeline. This setup is suitable when the sequence of jobs does not influence the outcome, as the actions performed by the pipeline do not impact other jobs. For instance, when a new change is submitted for review, tests run on this change provide immediate feedback. Since the change is not merged upon test completion, these tests can be executed in parallel with others without any interdependence. Additionally, a post-merge pipeline is considered independent because once changes are merged, the subsequent actions cannot affect any other jobs within the pipeline. Due to time constraints, developers typically put long-running tests in the post-merge pipeline because it would be impractical to include them in the pre-merge or merge queue pipeline. By running these extensive tests only in the post-merge pipeline, where the changes are already merged, broken tests are not immediately visible. To address these issues, our industry partner developed CTRA (CI Test Results Analysis), a tool that displays a timeline for each flaky and broken test case, depicting the results of its most recent executions, similar to the *Odeneye* system developed by Spotify [16].

Building upon understanding Zuul's CI system, exploring how Zuul jobs are defined and executed is essential. Ansible playbooks [4] define and configure Zuul jobs to automate workflows via YAML files. Ansible split the structure of playbooks into three phases: pre, main, and post, each serving a specific purpose in the job lifecycle (see Figure 4.2). The pre-phase set up the required environment for the job, such as cloning the source code from GitHub. This phase prepares the testing environment and ensures consistency for subsequent stages. The playbook's main phase is where the core action occurs. During this phase, tests are built and executed using Bazel [5].

Bazel is an open-source build and test tool similar to other engineering systems like CloudBuild [17] and Buck2 [6]. It utilizes a high-level programming language, enabling the creation of fast and reproducible builds. Bazel supports a wide variety of programming languages and platforms. Further, it offers a scalable architecture that extends its capabilities to suit various build scenarios for large and multi-language projects. The

[4] `https://www.ansible.com/`, visited on 2024-04-13
[5] `https://bazel.build/`, visited on 2024-04-13
[6] `https://buck2.build/`, visited on 2024-04-13

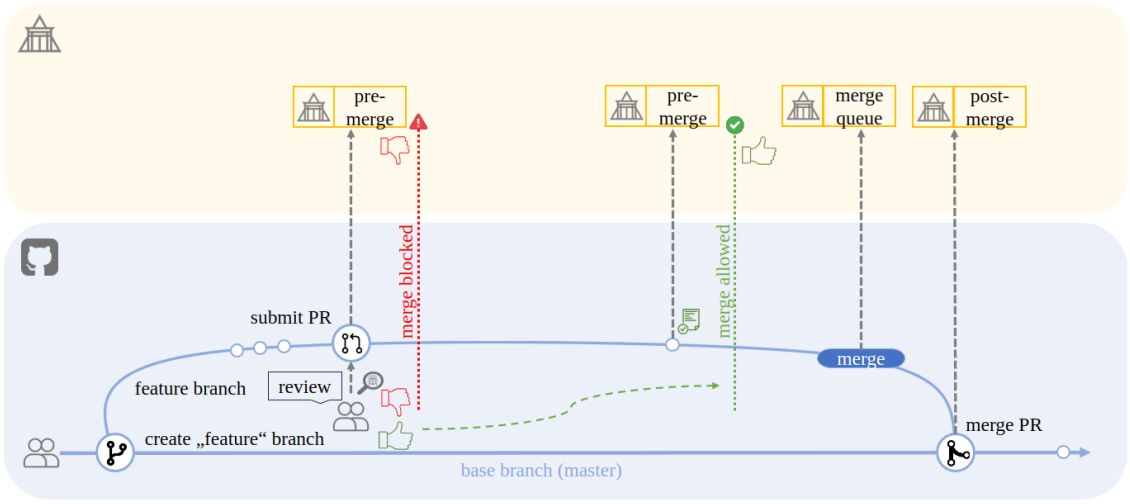

Figure 4.1: *The Continuous Integration workflow at BMW. Following the completion of a positive pre-merge check and review by a developer, the feature branch is merged into the base branch if the speculative merge with other changes is successful. Subsequently, the post-merge pipeline is executed on the new base branch.*

tool emphasizes the efficiency of incremental builds and tests. It uses a dependency graph, allowing it to selectively rebuild and test only those parts of the system affected by recent changes. This significantly reduces build and testing times, which is particularly beneficial in CI environments because development cycles and costs are decreasing [27]. Additionally, Bazel's sandboxing feature ensures that the outputs of builds and tests are consistent and unaffected by the system's state, enhancing the process's reliability.

4.1.2 Flakiness Detection

The issue of flakiness represents a significant challenge within CI systems [42], as tests are executed with greater frequency, thereby exposing flakiness more often [30]. When a flaky test failure occurs, it has the potential to impact the result of the entire build, which could ultimately threaten the overall efficiency of the CI process [12]. To address the issue of flakiness within the CI process, BMW re-executes tests on an initial failure. Other large software companies, such as Google [39] and Microsoft [32], use a similar approach to manifest flaky test failures. BMW leverages the built-in Bazel feature, which automatically reruns the test up to two additional times if the initial run fails. This helps to confirm whether the failure was indeed flaky, thereby mitigating the disruptive impact of flakiness on the CI efficiency. Nevertheless, only test cases that are identified as flaky during the nightly rerunning process are executed again.

We then utilize the data from the rerun mechanism to investigate flaky failures.

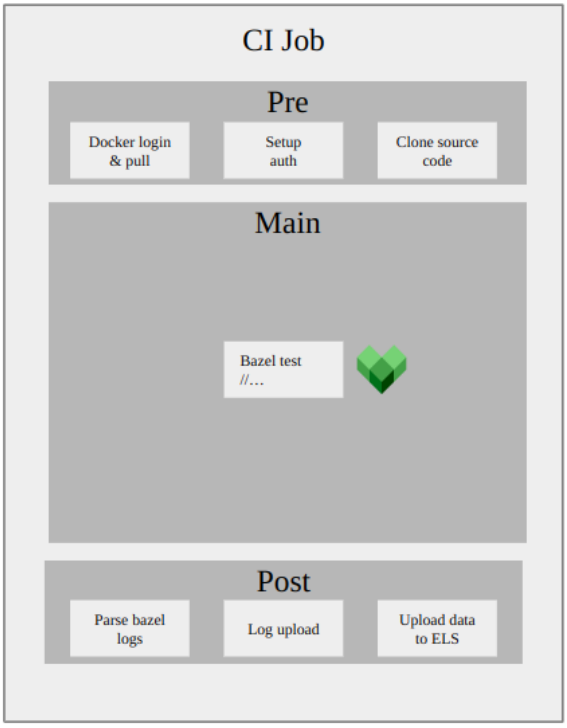

Figure 4.2: *CI job phases.*

4.2 Data Collection

After the main phase of each Zuul playbook, an automated process collects data from both Zuul and Bazel. The data is then written into an Elasticsearch [7] index and saved for six months. This integration ensures that all relevant test execution data is captured and stored in a centralized location for subsequent analysis. Additionally, Zuul saves the Bazel log output, which can be critical for diagnosing issues and understanding test behaviors. These logs are accessible through a URL associated with an entry in the Elasticsearch index, which includes test outcomes and detailed logs. However, these logs are only available for two weeks.

Before the actual data collection, we conducted a preliminary overview to determine the relevance of the data available. First, we discovered that a significant portion of the test results, denoted by 91%, were CACHED PASSED due to Bazel's caching capabilities. These tests were not executed and were thus excluded from our study, as they do not provide new insights into flakiness. Further investigation revealed that, although our project consisted of 38 submodules, we only had flaky failure data from 25 of them. Across these submodules, the number of test executions ranged from $\sim$ 200 to $\sim$ 64 Million within six months, excluding CACHED PASSED ones. For our study, we focused on submodules with more than 4 million test executions, narrowing our scope to 8

[7] `https://www.elastic.co/de/elasticsearch`, visited on 2024- 04-13

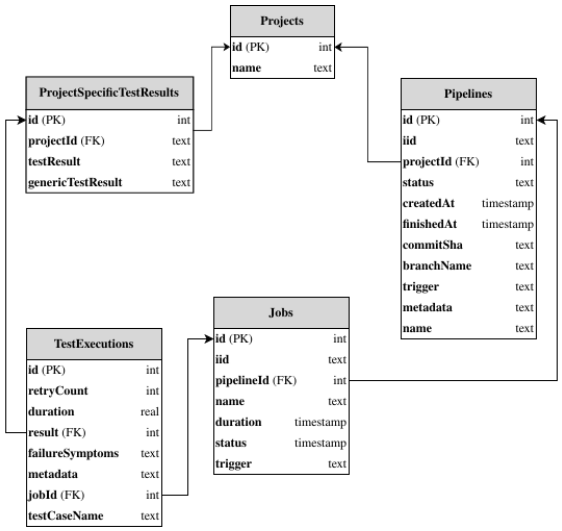

Figure 4.3: *Database schema with primary (PK) and foreign keys (FK).*

submodules. However, it is essential to note that the submodule representation in the data does not necessarily indicate that the test resides within that submodule. Instead, it signifies that a change was made in the submodule.

Given the monorepo structure of our codebase, there could be dependencies between submodules that affects which test is executed. Despite this, we used the data from these eight submodules as a filter, operating under the assumption that it is highly probable for most tests executed to be related to the submodule in which the change occurred. Upon an initial examination of the data and after applying some filters, we delved into the test results to better understand their outcomes. The data encompassed a variety of test results, including `FAILED`, `FAILED TO BUILD`, `FLAKY`, `NO STATUS`, `PASSED`, `SKIPPED`, and `TIMEOUT`. For our study, we consciously decided to only consider test executions that resulted in `FAILED`, `FLAKY`, or `PASSED`. The rationale behind this selection was that other results, such as `FAILED TO BUILD`, `NO STATUS`, and `SKIPPED`, did not provide any insight into test flakiness since the tests themselves were not executed. Although tests with the `TIMEOUT` result were indeed executed, they were terminated as Bazel enforces a time limit for each test based on its specified timeout attribute. These `TIMEOUT` results do not offer valuable data for our study of flakiness, as they represent a different kind of issue related to test execution time rather than non-deterministic test behavior. Consequently, we excluded `TIMEOUT` results from our analysis to maintain a clear focus on understanding and characterizing the flakiness of tests within our CI system. This approach allowed us to refine our data collection process and focus on the most significant subsets of data for analyzing test flakiness.

We developed an automated data collection process to fetch data daily from the Elasticsearch index. As mentioned before, filters established during preliminary research guide the data retrieval. This ensures that only the most relevant data is collected for analysis. In addition to this, the process involves downloading Bazel logs to automatically extract failure symptoms. The level of detail in these logs is configurable through

Bazel's test_output flag, which determines the granularity of the log information. When failure symptoms are detected, they typically consist of the combined standard output (stdout) and standard error (stderr) from the failed tests. Since the data is only accessible temporarily, it is crucial to store it in a permanent database. For this purpose, the data is saved into a SQLite database. This database is structured to generalize the data representation, allowing for the inclusion of data from various sources, such as different open-source projects or datasets from other companies. The database (Figure 4.3) contain the following five tables:

Projects: The Projects table represents the different projects from which data is collected. Each project in software development often has unique test results, which can vary significantly in terminology and meaning.

ProjectSpecificTestResult: The database schema includes a ProjectSpecificTestResult table to harmonize the diversity of different project-specific test results and enable standardized analysis across various projects. This table is instrumental in mapping project-specific test results to a more generic framework of understanding. The ProjectSpecificTestResult table acts as a translation layer, taking the many possible outcomes specific to each project and aligning them with two broad, generic test result categories: 'fail-like' and 'pass-like'. Doing so simplifies the complexity of interpreting test outcomes from different projects, allowing for a unified approach to assessing test success or failure.

Pipelines: The Pipelines table captures the details of each pipeline execution in a row. The `status` column within this table indicates the overall success or failure of the pipeline execution, providing immediate insight into whether the execution proceeded as expected or encountered issues. The columns `createdAt` and `finishedAt` record the start and end timestamps, respectively, providing precise temporal boundaries for each pipeline execution. The `commitSha` represents a specific moment in the code's history. As pipeline executions in Zuul are based on pull requests, we store a link to the pull request in Github. This field effectively ties the pipeline execution back to the corresponding code changes, facilitating traceability. The column `branchName` displays the name of the base branch from which the feature branch is checked out, providing context for the pipeline's execution environment. The `trigger` attribute captures the reason for the pipeline's execution, such as a manual trigger or a comment in a pull request. However, this field remains empty due to limitations in the data provided by the Elasticsearch index, as the trigger information is not stored there. A `metadata` field is included to capture diverse data that may not conform to our schema but is nonetheless valuable for analysis. The pipeline's name is recorded to identify which pipeline was executed, whether a pre-merge, merge queue, or post-merge pipeline. This classification is important for comprehending the context of pipeline execution and for subsequent analyses that may focus on specific pipeline types within the CI process. Finally, the `projectId` serves as an identifier for the project in which the pipeline was executed.

Jobs: Building upon the intricacies of the pipelines table, the database schema further extends to encapsulate the granularity of execution at the job level through the Jobs table. Each pipeline execution is composed of one or more job executions, which are recorded in this table. To ensure seamless integration and data retrieval, each job execution is assigned a unique identifier (iid). This identifier is crucial as it provides a direct link to access corresponding data within the Elasticsearch index, allowing for detailed analysis and cross-referencing of job-specific information. The `duration` column within the Jobs table is particularly informative, detailing the exact time each job took to execute. This metric is vital for performance analysis and identifying potential bottlenecks within the job execution process. The `name` column refers to the playbook utilized during the job execution, clearly indicating the tasks and processes carried out. A `status` column is also present, which shows the outcome of the job run—whether it was successful or not. It is important to note that within this CI setup, the `trigger` column for the Jobs table remains null because, in the system, only the pipeline execution is directly initiated by specific events, such as pull request actions. Consequently, all jobs associated with a given pipeline and project are automatically executed without individual triggers.

TestExecutions: Regarding the database schema, the TestExecutions table is crucial for recording the outcomes of individual tests within each job execution. This table is designed to capture each specific test execution associated with a job. In cases where test executions are repeated, such as flaky tests, the Elasticsearch index stores these repetitions as a single entry. However, our database schema includes a `retryCount` column, for a more detailed view. This column provides precise information on the number of times a test was retried and allows us to split these repeated executions into individual entries, ensuring that each rerun is distinctly captured and stored. The combination of `testCaseName`, `jobId`, and `retryCount` is unique. Table 4.1 shows the difference between the Elasticsearch index (a) and how it is then stored in our database schema (b). Each test case is executed in the same job. TestCaseC has a flaky test result and has a `retryCount` of one. In our database schema, this flaky test execution is split into two executions, where the first is fail-like and the second pass-like. Furthermore, the TestExecutions table records additional information, such as the duration of each atomic test execution, providing insight into the time efficiency of the tests. Similar to the Pipelines table, a metadata field is also available to store supplementary data in a JSON-like format, which helps capture project-specific details or additional context about the test execution. The `failureSymptoms` column is included to document any issues encountered during failed or flaky test executions. The `testCaseName` field identifies the specific test case being executed. In contrast, the `jobId` is a foreign key that links back to the corresponding job in the Jobs table. The `result` column also acts as a foreign key, linking to a `genericTestResult` that categorizes the outcome of the test case.

To facilitate deeper analysis, we define a *test execution batch* as a unique pairing of a `testCaseName` and `jobId`. A test execution batch has the potential to be flaky, as it may encompass varying outcomes. Table 4.2b illustrates this concept with four test execution batches. Each batch is represented by a single test execution, except for `TestCaseC`, which

Table 4.1: *Simplified test executions get from Elasticsearch (a) and stored as atomic test executions in the database (b)*

(a)

testCaseName	jobId	retryCount	result
TestCaseA	1	0	PASSED
TestCaseB	1	0	PASSED
TestCaseC	1	1	FLAKY
TestCaseD	1	0	FAILED

(b)

testCaseName	jobId	retryCount	result
TestCaseA	1	0	pass-like
TestCaseB	1	0	fail-like
TestCaseC	1	0	fail-like
TestCaseC	1	1	pass-like
TestCaseD	1	0	fail-like

comprises two test executions. The flakiness of `TestCaseC` is evidenced by its mixed results: one test execution yields a passing outcome while the other results in a failure.

4.3 Dataset

From mid-February to mid-April 2024, we collected data from 81,829 pipeline executions (pre-merge, merge queue, and post-merge). Inside these pipelines, 86 distinct jobs were executed 253,476 times. A total of 19,594,834 test execution batches and 19,688,994 atomic test executions were collected, which are distributed over 27,245 unique test cases. From these unique test cases, 2,879 included at least one rerunning test execution batch and can be further broken down into 714, which exhibit flaky behavior. Additionally, from all fail-like atomic test executions, we collected 347,892 failure symptoms.

4.4 Abstraction of Failure Symptoms

We utilize failure symptoms collected from flaky test results to gain insights into the underlying causes of flaky failures and identify where the flakiness originates. Furthermore, the failure symptoms are a foundation for interviewing experts and obtaining more detailed information about flakiness in specific test cases.

A recent study indicates that developers use logs to determine whether a failure is flaky or not [23]. However, manually comparing failure logs becomes impractical as the number of failures increases in a CI system. This process is time-consuming and becomes less feasible with the large amounts of data produced by automated tests. To improve

this process, we abstract and cluster the failure symptoms. This involves simplifying the failure logs by focusing on the most essential features.

An et al. [4] explored methodologies for abstracting failure symptoms to facilitate a more streamlined analysis. Their research highlights two primary approaches: stack trace purification and number masking. Stack trace purification involves cleaning and standardizing stack traces to remove irrelevant or non-deterministic information, while number masking targets the removal or generalization of numerical values within the failure symptoms. They demonstrate that the application of number masking, based on anonymization [3], abstraction [41], removal of variables [26], and removal of numbers [47] can significantly reduce the diversity of failure symptoms, with a reduction of approximately 78% in the number of unique symptoms. This reduction is crucial for simplifying the process of identifying and grouping failure symptoms, which in turn aids in pinpointing the root causes of test failures more efficiently. Given the goal of decreasing the number of unique symptoms to investigate and consider the practical constraints of our testing environment, we have opted to employ number masking as our approach to abstracting failure symptoms. The rationale behind this decision is twofold. Firstly, the simplicity and effectiveness of number masking make it a compelling choice for our purpose: to utilize the abstract failure symptoms to inquire of the test developers about the test cases (see Section 4.5). Secondly, using various testing frameworks across different projects introduces an additional layer of complexity, where not all failure symptoms include a stack trace, which would render stack trace purification inapplicable in some instances. By implementing number masking, we can effectively generalize failure symptoms, stripping away the inconsequential variations and honing in on the core issues that need attention. This abstraction approach ensures a more manageable and focused analysis, allowing us to better group failure symptoms. As demonstrated by An et al. [4], we replaced all numbers with # as shown in an example in Figure 4.4. All hexadecimal numbers are masked using the regular expression 0[xX][0-9a-fA-F]+. In an iterative process and by manually examining the abstract failure symptoms among themselves, we found further attributes that may cause symptoms with the same error reason as different error messages. Table 4.3 lists the regular expressions we have

Table 4.3: *Regular expression used for failure symptom abstraction*

Regular Expression	Description
/tmp/[0-9a-zA-Z]+	Temp directory path
0[xX][0-9a-fA-F]+	Hex-Address
[a-fA-F\d]{32}	MD5-Hash
sha256:[a-fA-F0-9]{64}	SHA-Hash
\d+(\.\d+)?s \(\d+:\d{2}:\d{2}\)	Time
([1|2]\d{3})\-((0[1-9])|(1[0-2]))\-([0-2]\d|(3[0-1])) (([0-1]\d)|(2[0-3])):([0-5]\d):([0-5]\d).\d{3}	Datetime
((https?):((//)|(\\\\))+[\w\d:#@%/;$()~_?\+-=\\\.&]*)	URL-Address
={ 4,}	More than 4 equal sign
\d+	Numbers

employed to mask information irrelevant for identifying the root cause of a failure. We applied these regular expressions in the sequence in which they appear in the table. For example, one specific pattern we noticed was the use of equal signs appearing more

```
1 ================================== ERRORS ==================================
2 ______ ERROR at setup of test_whitelisted_apps_amsr_environmentvariables
      _______
3 target_config_fixture = {'ip_address': '160.48.199.34', 'ext_ip_address':
      '169.254.21.103', 'diagnostic_address': 34, 'serial_device': '/dev/t
      ...django_pkg': 'django_pkg.tar.gz', 'path_to_appliance': 'external/
      guestfs_appliance/file/downloaded', 'use_doip': False}
```

(a) *Example of an error message.*

```
1 = ERRORS =
2 ______ ERROR at setup of test_whitelisted_apps_amsr_environmentvariables
      _______
3 target_config_fixture = {'ip_address': '#.#.#.#', 'ext_ip_address': '
      #.#.#.#', 'diagnostic_address': #, 'serial_device': '/dev/t...
      django_pkg': 'django_pkg.tar.gz', 'path_to_appliance': 'external/
      guestfs_appliance/file/downloaded', 'use_doip': False}
```

(b) *Example of the abstract error message. The numbers are replaced with # through the process. More than 3 equal signs after each other are replaced with one equal sign.*

Figure 4.4: *Example of an error message before and after the abstraction.*

than three times in succession. We determined that in such cases, the equal signs often function as separators between different sections of an error message or indicate the beginning or end of the message. Therefore, we combined any series of more than three equal signs into one to simplify the error logs.

Furthermore, we masked SHA and MD5 hashes. These elements are frequently used as unique identifiers within specific contexts but do not contribute to understanding failure causes. We also found that time and datetime stamps do not provide insight into the cause of failures, so we masked these. Similarly, we identified URL addresses during our manual investigation as non-contributory to failure causes, and thus, we also masked them. By systematically applying these regular expressions, we were able to clean the failure logs of extraneous information, allowing us to focus on the data that is most pertinent to uncovering the root causes of flaky test failures.

4.5 Semi-structured Interviews

We employed semi-structured interviews as a methodological tool to address research questions RQ 1.2 to RQ 1.4 (root causes, origin, and types of tests having flaky failures). Our objective with these interviews was to delve deeper into the phenomenon of flaky failures and to gain a more nuanced understanding of the issue.

In the design and execution of our interviews, we followed the guidelines proposed by Hove and Anda [29]. In particular, we asked follow-up questions whenever possible, which allowed us to explore the interviewees' responses in greater depth. Moreover, we started each interview with questions about the context of the test cases to establish a general understanding before progressing to more specific inquiries.

We focused on the eight most common test cases where the most flaky failures occurred. In doing so, we achieve coverage of 82.1% of flaky failures in the data. This selection strategy enabled us to analyze a significant proportion of flaky failures while conducting a relatively small number of interviews. We identified the code owners for each test case through GitHub and contacted them to schedule interviews using Microsoft Teams. The invitations we sent included information about the background of the study, the definition of flaky failures, and their significance in software development. Furthermore, we included the interview questions in the invitation to provide participants with an overview of what to expect.

On average, we invited four people to each interview, although not all invitees would necessarily accept. In some cases, experts who received an invitation forwarded the interview to a colleague they felt was more suitable. Typically, two people accepted the invitation to participate. Table 4.4 shows an overview of the interview partners.

Since some experts were code owners for multiple test cases, we reduced the number of interviews from eight to seven. Each interview lasted between 15 and 30 minutes and

Table 4.4: *Overview Interview Partner (IP)*

Interview	IP	Area of responsibility
1	IP1	Product Development Autonomous Driving
1	IP2	Virtual protection (Driving Level 3)
2	IP3	Service Motion Planning
3	IP4	Protocol Data Unit Tunneling
3	IP5	Middleware and Communication
4	IP6	Reprocessing & Simulation Framework development
4	IP7	Reprocessing & Simulation Framework development
5	IP8	Autonomous Driving Function A
5	IP9	Autonomous Driving Function A
6	IP10	ECU operating system and integration components
7	IP11	Software development - Autonomous Driving

was conducted as a video conference through Microsoft Teams.

We conducted the interviews with a pair of interviewers fulfilling two distinct roles. One interviewer was responsible for asking questions and leading the conversation, while the other took detailed notes, capturing the responses provided by the interviewees.

The semi-structured interviews were guided by the questions outlined in Table 4.5, which served as a foundation for the discussion and ensured we covered all relevant aspects of the research topic.

Questions 1 and 2 were crafted to be open-ended and general, allowing the interviewees to provide a broad overview of the test case. On the other hand, questions 3 to 5 were much more targeted towards the issue of flaky failures. We based these questions on the abstracted failure symptoms outlined in Section 4.4 to ensure the interviewees could provide detailed and relevant responses. This allowed the participants to reflect on the specific flaky failures identified in our data.

Table 4.5: *Interview questions with answer options*

Question	Answer Options
Q1: Which function(s) is/are being tested by this test?	No answer options
Q2: Were you aware that this test was flaky prior to the meeting?	No answer options
Q3: What type of test is this?	- Unit test - Integration test - System test
Q4: What is the root cause of the flaky failure?	- Async Wait - Concurrency - Resource leak - Network - Time - IO - Randomness - Floating point operations - Unordered collections
Q5: Can the flakiness be localized and attributed to one of the following areas?	- Test - Code under Test - System under Test

Additionally, we aimed to standardize the responses from the interviewees by comparing them with predefined choices from existing literature. This helped to categorize the answers and facilitated the analysis of the interviews.

For Question 3, which inquired about the types of tests, we presented the interviewees with a set of potential choices, including Unit test, Integration test, and System test, as referenced in the literature [6, 28].

In Question 4, we focused on identifying the root causes of flaky failures, for which we utilized the categories defined by Luo et al. [38]. This provided a framework for interviewees to classify the causes they encountered in their experiences.

Question 5 was designed to ascertain the origin of the flakiness. We were particularly interested in whether the flakiness stemmed from the test itself or the CUT. Recognizing that tests could also be run on real hardware such as ECUs, we included System under Test (SUT) as another potential source of flakiness. Moreover, considering the context of CI environments, we added infrastructure as a possible origin of flakiness.

5 Study Results

This chapter presents the empirical findings of the research on flaky test failures within the CI environment. The data collected and analyzed from the CI system provide insights into the nature and behavior of flaky test failures. The results are organized to reflect the research questions and will delve into the quantitative results obtained from the study, providing a comprehensive overview of the patterns observed.

The detailed interview notes can be found in the appendix A.

5.1 Flakiness in the CI System

In total, we found that 99.72% of all test execution batches in the data did not have automatic reruns, which can be distributed to 19,292,215 passed and 247,504 failed test execution batches. That a test case is not automatically rerun on failure may be attributed to the test passing at the first execution or the test case not being identified as flaky by the automatic nightly flakiness detection in the CI. No reruns are initiated with the latter, regardless of whether the test passes or fails. This data is irrelevant to investigating the flakiness of the CI system, despite all failures potentially exhibiting flakiness if executed multiple times. The remaining 0.28% (55,115) of the test execution batches failed at the first execution and were triggered by the automatic rerun mechanism. This value can be further broken down into 0.18% (35,065) of tests that failed all subsequent reruns and 0.10% (22,006) of flaky failures. This results in a continual rate of 0.10% of test execution batches reporting a flaky result. The flaky failures are distributed across 714 unique tests, representing a ratio of 2.6% of all tests investigated.

RQ1.1 Which Share of Flaky Test Cases Is Responsible for a Certain Share of Flaky Failures?

By calculating the flake rate, which is the number of times the flaky test has a flaky verdict out of the total number of times the test was run, we find that the mean flake rate is 0.83%, the median 0.084%, the lower quartile 0.029%, the upper quartile 0.43%, and the maximum 93.46% over the entire two months. This indicates that most test cases exhibiting flaky behavior have a relatively small number of flaky test execution batches.

To ascertain the proportion of flaky test cases responsible for a given number of flaky failures, we examined the ten test cases with the most flaky test execution batches. As illustrated in Figure 5.1, these ten test cases account for approximately 82% of all flaky failures. Furthermore, the data indicates that three test cases are responsible for approximately 80% of the flaky failures. This result can be further broken down into 66%, 8.7%, and 5.2%, demonstrating that two-thirds of the flaky failures originate from one test case. The remaining 711 flaky tests have 5.75 flaky test execution batches on average and 3 on median. Consequently, they contribute only a minimal number of instances of flaky test failures.

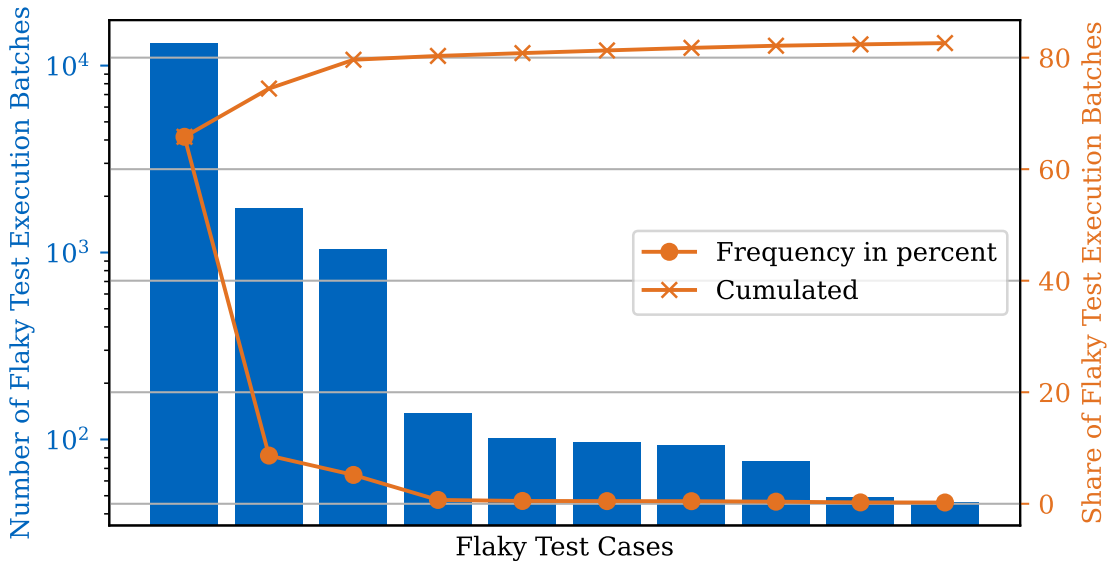

Figure 5.1: *Top ten flaky test cases with the most flaky test execution batches.*

RQ1.2 What Are the Root Causes of Flaky Failures?

Five of the study's participants (IP1, IP2, IP3, IP6, IP7) identified the CI system as the source of test flakiness. They pointed out that tests run in parallel within the CI environment, resulting in a higher workload than if they were run locally, which can contribute to flakiness. IP6 and IP7 described that an application can not be built within a defined time frame, which led the interviewees to identify the CI as the root cause. Additionally, one participant highlighted that flaky behavior in the CI could be traced back to inadequate dependency management, as suggested by the error message. In an interesting contrast, IP4 suggested that there is a tendency among developers to blame the CI for issues that stem from their code. He mentioned that replicating flaky behavior requires putting the system under stress, similar to the high-load conditions experienced by the CI.

RQ1.3 Where Do the Flaky Failures Originate From?

Five developers mentioned that the test itself is the origin of the flakiness (IP4, IP5, IP8, IP9, IP11). For instance, IP4 explained that the test is inherently challenging to develop and susceptible to flake due to the complexity of the interface through which it communicates. Flakiness that originates from the part of the system that is directly under test was mentioned by none of the developers. This indicates that product reliability is not directly affected by the flakiness we asked developers about. The System under Test was mentioned as the origin of the flakiness by one expert (IP10). Differently from the CUT, this category considers the system as a whole, not only the part under test. The SUT emerges as a source of flakiness in a complex system where the integration tests flake due to testing requirements on real hardware (ECU) not complying.

The testing infrastructure is the set of processes that support the testing activity and ensure its stability. Five interviewees considered their tests flaky because of an unstable or improper testing infrastructure (IP1, IP2, IP3, IP6, IP7). For instance, IP1 explained that the test case we asked about could be flaky because of CPU availability or too many parallel test executions on the CI node.

Other experts could not provide exact information about the origin of the test case. This was the case when, for example, the failure symptoms were not sufficiently meaningful or there was no investigation into the flakiness from the developer.

Table 5.1 illustrates the type of test from which the flaky failures originate and the number of test cases responsible for the flaky failures. In total, 69% of the flaky failures originate from the infrastructure, 4.7% of the SUT, 0.86% from the test itself, and 0% from the CUT. However, the percentages do not add up to 100% because only the statements of the interviewed flaky test cases are available.

Table 5.1: *Origin of flaky failures for interviewed test cases. Cells: Number of flaky failures (number of flaky test cases)*

Test Type	Origin							
	Test		CUT		SUT		Infrastructure	
Unit	45	(1)	0	(0)	0	(0)	0	(0)
Component	49	(1)	0	(0)	0	(0)	0	(0)
Integration	0	(0)	0	(0)	1037	(1)	15166	(4)
System	96	(1)	0	(0)	0	(0)	0	(0)

RQ1.4 What Type of Tests Are Responsible for Flaky Failures?

Developers classified six of the eight test cases as integration or system tests responsible for 17,299 flaky failures. This indicates that most flaky failures are associated with higher-order tests involving multiple testing units. Additionally, the flaky failures from one test case originated from a component test. IP4 describes a component test as a test focusing on testing a complete component in isolation from the entire system to verify its functionality. This is distinct from integration tests requiring interaction with other components or the entire system. A single flaky test case was traced back to a unit test.

Summary ***(RQ1: Flakiness occurence in this CI system)***
Within the CI setting, flakiness is present in 2.6% of test cases. These cases have resulted in 22,006 instances of flaky failures, accounting for 0.10% of the total test execution batches. Three test cases alone are the source of approximately 80% of these flaky failures. Three developers attribute the cause of flakiness to the high workload on the CI system. Furthermore, the test code owners point out that 69% of the flaky failures are due to issues within the infrastructure. Additionally, they note that seven out of eight tests are component, integration, and system tests, which means they cover more than just an isolated unit, potentially increasing the complexity of diagnosing and addressing flakiness.

5.2 Surviving of Flaky Failures After Rerunning

We used an iterative approach to study the phenomenon of failures that *survived* reruns and, instead of showing up as flaky, showed up as failures in all three executions.

1) **Intersection**: Initially, we examined the 714 test cases already identified as flaky and cross-referenced them with the 2,737 unique test cases that have a test execution batch that failed all three reruns at least once. Next, we performed an intersection analysis

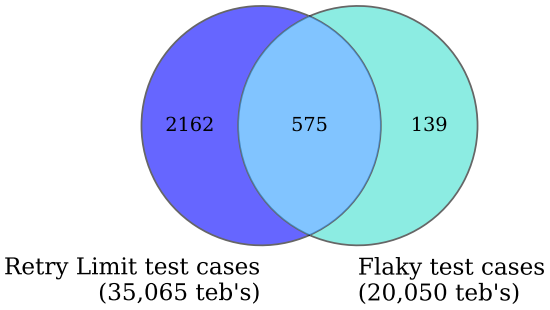

Figure 5.2: *Intersection of test cases with flaky and test execution batches (teb's) reaching the retry limit.*

between the flaky test cases and those that reached the retry limit (Figure 5.2). Our findings revealed that 575 (81%) of the flaky test cases also had test execution batches that failed three consecutive times. This analysis led us to identify 13,172 failed test execution batches that are potentially flaky but would require more than three reruns to be confirmed as such.

2) **Flakiness time frame**: To enhance the likelihood that a retry limit failure is indeed

Figure 5.3: *Example timeline of a test case with its first and last flaky result in the data. Only use retry limit failures in between this range.*

flaky, we excluded test execution batches that did not occur within the same time frame as the identified flakiness of a test (see Figure 5.3). This refined our data to 336 flaky test cases and 5,808 potential test execution batches that were incorrectly identified as regressions. Of these test execution batches, 4,609 exhibited observable failure symptoms.

3) **Comparison of Failure Symptoms**: In the final phase of our analysis, we compared the failure symptoms of these test execution batches with those from flaky test executions for the same test cases. Where there was a match in the abstract failure symptoms, we classified the failures as *survived* flaky failures.

Through this process, we discovered that 205 test execution batches could be flaky despite failing all three reruns. These batches were spread across 60 different test cases. Resulting in 10% of the test cases having at least one potential *survived* flaky failure. In total, 4.5% of the test execution batches, which were retry limit failures conducted within the same period that a test case exhibited flakiness and had failure symptoms, could be considered potential *survived* flaky failures.

Summary *(RQ2: Surviving of flaky failures)*
We analyzed 714 flaky test cases and identified 575 (81%) that also had at least one test execution batch that failed for all three reruns. After excluding non-relevant data, we narrowed potential flaky failures to 205 test execution batches across 60 test cases. In total, 10% of test cases exhibited at least one potential flaky failure, and 4.5% of test execution batches could be considered *survived* flaky failures despite failing all rerun attempts.

5.3 Effectiveness of Rerunning Tests

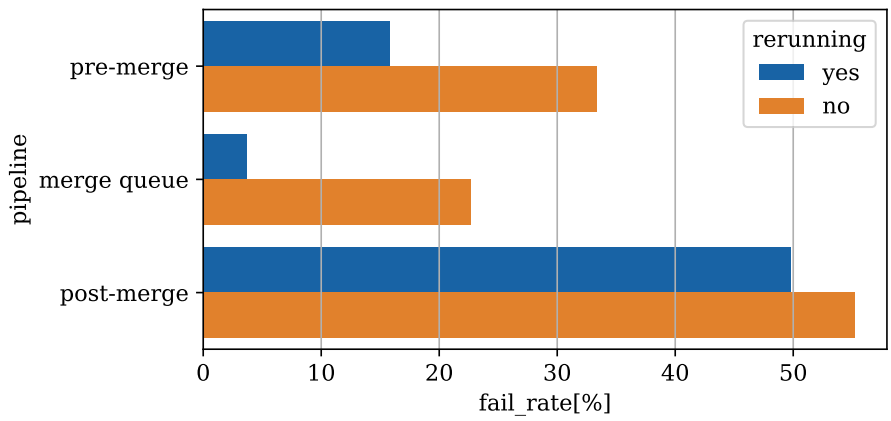

Figure 5.4: *Fail rate of pipelines with and without rerunning.*

To answer RQ3, we determine the significance of rerunning by assessing the potential increase in failures if rerunning was not implemented. We analyzed the outcomes of pipelines, focusing on those that succeeded, to understand the impact of rerunning on

the stability of the CI process. Figure 5.4 compares the pipeline failure rates with and without the practice of rerunning tests after a initial failure. Our analysis indicated that the most notable differences in failure rates due to rerunning were observed in the pre-merge and merge queue pipelines. The merge queue pipeline, in particular, exhibited the lowest failure rate when rerunning was applied. This is a crucial finding because a single job failure in the merge queue pipeline can trigger a reset and rerun of the entire merge queue, which is both time-intensive and expensive. In scenarios without rerunning, the failure rate in the merge queue pipeline would slightly exceed 20%. The post-merge pipeline displayed the highest failure rate in nearly 50% of cases, even when rerunning was in place. One possible explanation for this high failure rate is that tests with a history of flaky results in the pre-merge and merge queue pipelines are often shifted to the post-merge pipeline.

Figure 5.5 displays the number of reruns each test uses to pass after an initial failed

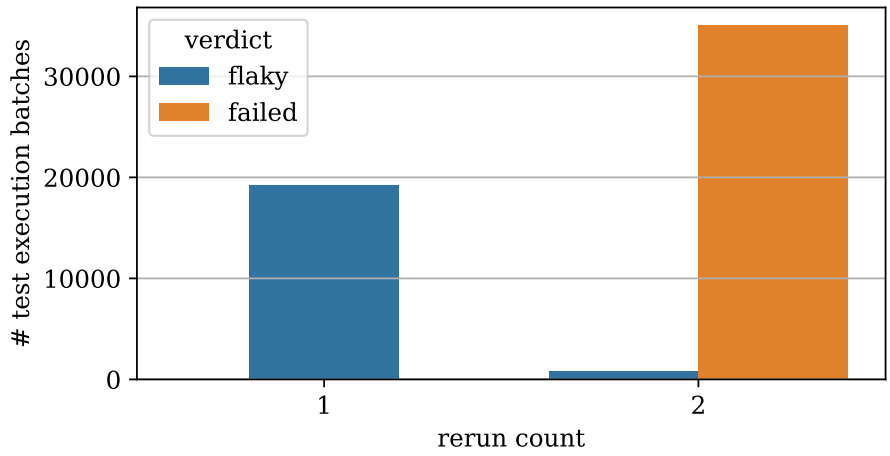

Figure 5.5: *Number of test execution batches for rerun counts one and two after failure, differentiated by the test verdict.*

result. 96% of these batches pass on the first rerun. A smaller proportion of flaky test execution batches require a second rerun to pass. This outcome indicates that the mechanism currently used to determine whether a test case should be rerun, based on nightly execution data, effectively detects flaky test cases most likely to be resolved after the first rerun. All failed test execution batches are rerun twice, as this is the maximum number of reruns allowed following an initial failure. When focusing solely on the flaky test execution batches, increasing the number of allowed rerun attempts would not incur additional costs. However, as addressed in RQ2, the likelihood that a test execution batch is flaky after failing three consecutive times is approximately 4.5%. Therefore, increasing the rerun limit could identify additional flaky tests.

Conversely, there is a risk that increasing the rerun attempts may lead to failures that persist beyond three reruns without yielding new insights. This would extend the execution time, delay the feedback to developers, and incur higher costs due to more reruns. Thus, while there may be benefits to increasing rerun attempts for identifying

flaky test failures, it must be balanced against the potential for longer execution times and higher CI costs.

Summary ***(RQ3: Effectiveness of rerunning tests)***
The study for RQ3 evaluated the significance of rerunning tests in CI pipelines, finding that it significantly reduces failure rates, particularly in pre-merge and merge queue pipelines. Failure rates would be notably higher without rerunning, e.g., the merge queue pipeline exceeding a 20% failure rate. Despite rerunning, post-merge pipelines still showed high failure rates, possibly due to the relocation of unstable tests from the merge queue to the post-merge pipeline.
Analysis showed that 96% of tests pass after the first rerun. While increasing the number of reruns could identify more flaky tests, it also risks longer execution times and higher costs.

6 Discussion

Flaky failures are concentrated in a few test cases. When examining the flakiness within the CI system, we found that only 2.6% of the test cases exhibited flakiness. This could be attributed to the fact that only a minority of test cases are selected for rerun after an initial failure. Consequently, other tests could be flaky but go undetected due to the lack of rerunning failed executions. However, the criteria for rerunning a test case are based on data from multiple nightly executions. This lends credibility to the belief that the number of flaky test cases identified is realistic. Moreover, this result is consistent with findings from other CI systems, such as those used by Google [39] or Microsoft [32]. Google mentioned that 16% of all tests are flaky and over six Microsoft project 3.8% flaky test show up. These numbers support the assumption that only a few test cases show flaky behaivor in a CI system.

A more detailed analysis of the flaky test cases revealed that a limited number are responsible for most of the flaky failures. This shows that not every flaky test case has the same impact on the CI system, contradicting the presumption made by previous research [21, 31, 38]. Given that flakiness is a significant challenge in CI systems [12, 20], a practical recommendation would be to prioritize fixing test cases with higher failure rate, which could significantly reduce the occurrence frequency of flaky failures. More specifically, in the CI system that we examined, fixing the single test case with the most flaky failures would reduce the total number of flaky failures by 66%.

Survived flaky failures as a relevant research problem. Flaky test failures that are identified by the rerunning mechanism do not affect developers directly as the pipeline continues to pass. However, we found that some flaky failures *survive* reruns and therefore still manage to impact the pipeline in which they are executed. An analysis of all failures that also failed during rerunning within the CI system has revealed that approximately 4.5% of test failures are *survived* flaky failures. This figure is considered a lower limit estimate due to the conventional approach used to identify *survived* flaky failures, because the failure symptoms of flaky failures and retry limit failures must be the same. Consequently, only those test cases with exactly the same abstract failure symptoms were included in the evaluation. It is plausible that even not exactly equal failure symptoms are indicating that the same reason a behind the error. Further research with a more sophisticated approach could reveal a higher incidence of *survived* flaky failures and can provide a more comprehensive understanding of the prevalence of *survived* failures within the CI system.

Larger tests tend to be more flaky. The developers we interviewed for this study indicated that test cases exhibiting a high number of flaky failures are typically those testing multiple units together or the complete system (integration-, system-, component-test). The answer show that the there is a link between the size of a test and the likelihood of a test being flaky. Listfield [37] mentioned that the likelihood of a test being flaky

increases as the test size increases, what is consistent with our statement. They supported this claim by using the binary size and the used RAM to quantify the test size. To strengthen our statement, it is recommended that the same metrics should be collected in future studies. Moreover, as one developer posited, the complexity of the CUT may be a contributing factor to the flakiness of a test case. Therefore, it would be beneficial in future work to employ and analyze different metrics for the complexity of the test code and the tested code to ascertain whether there is a correlation with flakiness.

Rerunning test failures lowers the fail rate of CI pipeline substantially. The results of the analysis show that the fail rate in the post-merge pipeline is many times higher than in the merge queue pipeline regardless of whether rerunning is active or not. This is because CI developer focus on keeping the failure rate in the merge queue as low as possible. To achieve this, an automatic process is run which analyzes the test results of the merge queue over the last 24 hours. If a test is already marked as flaky and fails despite the re-run mechanism, it is marked as unstable and moved to the post-merge queue. This means that a change will now be merged irrelevant if the test fails or passes. This could explain the higher failure rate observed in the post-merge pipeline, as the failing tests are effectively moved out of the merge queue. We speculate that in such cases, developers may not always investigate the test failures because they do not realize that the test is failing. The pre-merge pipeline, on the other hand, is different in this regard, since the change cannot be merged without passing all tests. Therefore, any regressions must be fixed by the developers before the change can be merged.

7 Threats to Validity

External Threats: The nature of the data employed primarily constrains the study's external validity. The dataset is derived from a single, large-scale industrial software project, which inherently limits the generalizability of the results. The CI system in question, Zuul with the build system Bazel, are specific tools with particular characteristics and may not represent all CI systems in use across various software development environments.

The observed flakiness within the CI system has been compared with findings from other studies. However, these comparisons are constrained by the unique aspects of the dataset and CI systems employed in this research. While the results are valuable, they are specific to the context of the single project and the CI tools analyzed.

Further evaluations involving more diverse datasets are necessary to enhance these findings' generalizability. These datasets should encompass multiple proprietary and open-source projects with varying scales and complexities. Furthermore, examining different CI setups would be beneficial to understanding how flakiness manifests across various systems and configurations. Only through such expanded research can the results be confidently generalized to a broader range of software development practices and CI system implementations.

Internal Threats: This study has several internal threats. While there is evidence to suggest that the test cases we consider to be flaky are indeed flaky, some of the test cases we consider to be non-flaky may be flaky. However, the process of detecting flaky test cases is robust, as it depends on two key factors: (1) the nightly execution of test cases, which produces data that is automatically analyzed, and (2) the expertise of developers, who are responsible for manually confirming these results. In addition to the process that is designed to identify flaky test cases and rerun them in the event of failure, there is a possibility that a test case that is initially identified as flaky may not exhibit such behavior throughout the two-month data collection period. As a result, the test may be misclassified as non-flaky.

Another threat to internal validity is the possibility that the participants invited to the interviews to answer questions about flakiness (origin and root cause of flakiness and which type of test is responsible) may not be the same individuals who initially wrote the test case due to a lack of availability resulting in the company or another company that was commissioned to write the test being represented. This could lead to misunderstanding the interview questions and a lack of expertise. However, only participants who are listed as code owners were utilized, which implies specific expertise, regardless of whether they wrote the test or not. Furthermore, the interviews were not recorded or transcribed, which could result in aspects of the participants' statements being overlooked. To mitigate this threat, one of the two individuals was primarily responsible for taking notes during the interviews.

8 Conclusions

The phenomenon of flakiness represents a significant challenge in software engineering. While numerous studies have examined the prevalence of flaky tests, only a few have focused on the occurrence of flakiness within the dynamic context of a Continuous Integration (CI) environment, where the Code under Test (CUT), test code, and System under Test (SUT) are constantly evolving. Furthermore, the focus of researchers is on flaky test cases rather than flaky failures, which indirectly claims that every flaky test is equally problematic.

This case study at BMW empirically demonstrated that test cases do not have an even distribution of flaky failures. Three test cases are responsible for approximately 80% of all flaky failures. This contrasts with the assumption that every flaky test is equally problematic, which is frequently made by researchers. Interviews with experts revealed the following findings: (1) the CI is the primary root cause, (2) component, integration, and system tests are the most prevalent type of test associated with flaky failures, and (3) the infrastructure is the origin of the majority of flaky failures. Notably, we observed that 4.5% of the flaky test executions that fail continuously are potentially flaky after compared with the failure symptoms of existing flaky failures. This indicates that the current strategy of a fixed number of rerun attempts could be modified to a dynamic, adjustable version that considers historical data and failure symptoms of flaky failures. Rerunning test failures lowers the CI pipeline failure rate. In the merge queue, rerunning reduced the fail rate by up to 20%, in the post-merge by approximately 5%. The variation of the two values can be attributed to the different tasks the pipelines have in the CI system. Overall, however, the rerunning affects only a small portion of all test execution batches, amounting to 0.28%.

This study is limited by the fact that we analyzed only the data from our industrial partner. Despite this limitation, this study makes an important contribution because, to the best of our knowledge, we are the first to empirically analyze flaky failures rather than flaky tests in the context of CI. For future research, we recommend that others analyze flaky failures in open-source projects and in other proprietary projects that utilize a specific CI setup to reproduce our findings and generalize the results of this study. Therefore, we suggest employing our database schema for easy comparison.

A Interview Notes

Table A.1: *Notes of interview 1*

Q1: Which function(s) is/are being tested by this test?	- Closed loop simulaiton on artificial data - Checks if simulation starts - Only execute few steps
Q2: Were you aware that this test was flaky prior to the meeting?	- Did know that the test is sometimes flaky - They do not normally look at the test either - Generic Test, different costumer function can be tested - Another team is more responsible, maybe no one is interested in the test anymore
Q3: What type of test is this?	- Difficult to classify, possibly integration test because they integrate the Autonomous Driving Function A into the simulation
Q4: What is the root cause of the flaky failure?	- Could be because of CPU availability - Not possible to reproduce the flakiness locally, therefore hard to say what root cause - sometimes the test need more time in the CI and runs into a timeout - paralleslism could be the problem
Q5: Can the flakiness localized and attributed to one of the following areas?	- Infrastructure
Additional information	- They only do something if flakiness could be reproduced locally - We set the test up initially, but the Autonomous Driving Function A function are done by a different team - We sometimes also have flakiness inside, which we can't understand

Table A.2: *Notes of Interview 2*

Q1: Which function(s) is/are being tested by this test?	No answers were provided by the interview partners.
Q2: Were you aware that this test was flaky prior to the meeting?	- Was not aware that test is flaky, if they did know it they had did something
Q3: What type of test is this?	- Integration test
Q4: What is the root cause of the flaky failure?	- Regarding the error message a specific library could not be loaded - Maybe a library is not in every CI Node $\rightarrow$ dependency management - Depends on files that load a specific map or road data
Q5: Can the flakiness localized and attributed to one of the following areas?	- Infrastructure failure
Additional information	

Table A.3: *Notes of Interview 3*

Q1: Which function(s) is/are being tested by this test?	- multiple ECUs and sensors in a car - specific ECU wants the data - sensors don't have much communication ability → legacy sensors - newer ECUs don't have access do old bus systems - need for system that converts data that coming from bus system into data that can send via ethernet → the system is doing this - Test cover component that is doing network logic - testing that component is able to receive packages via udp and send them forward via ipc - in a virtual environment
Q2: Were you aware that this test was flaky prior to the meeting?	- attendants were aware that the test is flaky - because of high flake rate
Q3: What type of test is this?	- component test, one component without entire system - binary is using the real interfaces provided by the system, but the system's functionality is mocked
Q4: What is the root cause of the flaky failure?	- Not necessary network, because its running a very clean environment - Long tool chain, if same small issue - More of a problem is that component test is on such high level that we cannot look at every small piece
Q5: Can the flakiness localized and attributed to one of the following areas?	- Test code, because the interface we communicate via IPC is not the simplest one → making mistake is quite likely
Additional information	- CI is under high load → timing behavior changes → when reproducing a crash/failure on the CI, system needs to be under stress - Common misconception of developers around here, tend to push problems on the CI when their problems are the problemC - Chances that the CI has a problem that your local machine doesn't have are quite low

Table A.4: *Notes of Interview 4*

Q1: Which function(s) is/are being tested by this test?	- Uses final RPU (Reprocessing Unit) and executed some smoke tests - A whole docker container is tested - It is only tested, that the docker container as well as in and outputs are available
Q2: Were you aware that this test was flaky prior to the meeting?	- Yes. Why it has so many runs: ran on master and release branches (now no longer master) - Currently only runs on feature branch.
Q3: What type of test is this?	- Integration test
Q4: What is the root cause of the flaky failure?	- Normally the test doesn't fail locally, but with multiple runs it can be reproduced - In the CI the test is not running alone, higher workload - Docker container can not build within 60 sec - Possible fixes: increase timeout, don't allow that test runs in parallel
Q5: Can the flakiness localized and attributed to one of the following areas?	- Infrastructure, Time
Additional information	

Table A.5: *Notes of Interview 5*

Q1: Which function(s) is/are being tested by this test?	- Acceptence test - The road is made narrower, than the system has to be switched off
Q2: Were you aware that this test was flaky prior to the meeting?	- Get mail if test is flaky but ignoring them
Q3: What type of test is this?	- System test
Q4: What is the root cause of the flaky failure?	- Locally the tests are not failing (Why they sometimes pass in the CI, could not be answered) - Maybe something is not initialized right - Tests are not changed for the new automotive generation
Q5: Can the flakiness localized and attributed to one of the following areas?	- More likely test code
Additional information	- Seems that often there are test that someone war responsible for and is now gone. - Don't has time to analyse flakiness

Table A.6: *Notes of Interview 6*

Q1: Which function(s) is/are being tested by this test?	No answers were provided by the interview partners.
Q2: Were you aware that this test was flaky prior to the meeting?	- Yes, fixed flakiness in other branch but the cherry-pick is missing, so the changes who fixed the flakiness is not introduced in the specific branch
Q3: What type of test is this?	- Unit test
Q4: What is the root cause of the flaky failure?	- IO - The unit-test patches os.walk which isn't present any longer, therefore it dispatches to root-level and search the complete file-tree starting from root. In case the root contains files which matches somehow the files of the test-setup it could be fail arbitrarily
Q5: Can the flakiness localized and attributed to one of the following areas?	- Test
Additional information	

Table A.7: *Notes of Interview 7*

Q1: Which function(s) is/are being tested by this test?	- Test on real hardware
Q2: Were you aware that this test was flaky prior to the meeting?	- Yes
Q3: What type of test is this?	- Integration test
Q4: What is the root cause of the flaky failure?	- Not a network problem, we are abusing the network here
Q5: Can the flakiness localized and attributed to one of the following areas?	- System under Test - In engineering mode, additional logging → leads to flaky failures
Additional information	- New project, not looking into test anymore

Acronyms

CI Continuous Integration

SUT System under Test

CUT Code under Test

ECU Electronic Control Unit

LOC Lines of Code

List of Figures

List of Tables

Bibliography

[1] Amal Akli et al. "Flakycat: predicting flaky tests categories using few-shot learning". In: *International Conference on Automation of Software Test (AST@ICSE)*. 2023, pp. 140–151.

[2] Abdulrahman Alshammari et al. "230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers". In: *arXiv preprint arXiv:2401.15788* (2024).

[3] Anunay Amar and Peter C Rigby. "Mining historical test logs to predict bugs and localize faults in the test logs". In: *International Conference on Software Engineering (ICSE)*. 2019, pp. 140–151.

[4] Gabin An et al. "Just-in-time flaky test detection via abstracted failure symptom matching". In: *arXiv preprint arXiv:2310.06298* (2023).

[5] Morena Barboni, Antonia Bertolino, and Guglielmo De Angelis. "What we talk about when we talk about software test flakiness". In: *International Conference on the Quality of Information and Communications Technology*. 2021, pp. 29–39.

[6] Luciano Baresi and Mauro Pezzè. "An Introduction to Software Testing". In: *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT)*. 2004, pp. 89–111.

[7] Kent Beck. "Embracing change with extreme programming". In: *Computer* (1999), pp. 70–77.

[8] Jonathan Bell et al. "DeFlaker: Automatically detecting flaky tests". In: *International Conference on Software Engineering (ICSE)*. 2018, pp. 433–444.

[9] Grady Booch. *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., 1990.

[10] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. "T-Evos: A Large-Scale Longitudinal Study on CI Test Execution and Failure". In: *IEEE Transactions on Software Engineering* (2023), pp. 2352–2365.

[11] *Continuous Integration*. URL: `https://martinfowler.com/articles/continuousIntegration.html` (visited on 2024-03-17).

[12] Thomas Durieux et al. "Empirical Study of Restarted and Flaky Builds on Travis CI". In: *International Conference on Mining Software Repositories (MSR)*. 2020, pp. 254–264.

[13] Saikat Dutta, August Shi, and Sasa Misailovic. "Flex: fixing flaky tests in machine learning projects by updating assertion bounds". In: *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2021, pp. 603–614.

[14] Saikat Dutta et al. "Detecting flaky tests in probabilistic and machine learning applications". In: *ACM SIGSOFT International Workshop on Software Analytics*. 2020, pp. 211–224.

[15] Moritz Eck et al. "Understanding Flaky Tests: The Developer's Perspective". In: *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019, pp. 830–840.

[16] Spotify Engineering. *Test Flakiness - Methods for identifying and dealing with flaky tests*. 2019. URL: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/ (visited on 2024-04-13).

[17] Hamed Esfahani et al. "CloudBuild: Microsoft's distributed and caching build service". In: *International Conference on Software Engineering (ICSE)*. 2016, pp. 11–20.

[18] *Git Tools - Submodules*. URL: https://git-scm.com/book/en/v2/Git-Tools-Submodules.

[19] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. "On the rise and fall of CI services in GitHub". In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2022, pp. 662–672.

[20] Martin Gruber and Gordon Fraser. "A survey on how test flakiness affects developers and what support they need to address it". In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2022, pp. 82–92.

[21] Martin Gruber et al. "An Empirical Study of Flaky Tests in Python". In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2021, pp. 148–158.

[22] Martin Gruber et al. "Practical flaky test prediction using common code evolution and test history data". In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2023, pp. 210–221.

[23] Sarra Habchi et al. "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests". In: *International Conference on Software Testing, Verification and Validation (ICST)*, pp. 244–255.

[24] Mark Harman and Peter O'Hearn. "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis". In: *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2018, pp. 1–23.

[25] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. "An empirical study of flaky tests in javascript". In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2022, pp. 24–34.

[26] Pinjia He et al. "Drain: An online log parsing approach with fixed depth tree". In: *IEEE International Conference on Web Services (ICWS)*. 2017, pp. 33–40.

[27] Michael Hilton et al. "Usage, costs, and benefits of continuous integration in open-source projects". In: *International Conference on Automated Software Engineering (ASE)*. 2016, pp. 426–437.

[28] Itti Hooda and Rajender Singh Chhillar. "Software test process, testing types and techniques". In: *International Journal of Computer Applications* (2015).

[29] Siw Elisabeth Hove and Bente Anda. "Experiences from conducting semi-structured interviews in empirical software engineering research". In: *IEEE International Software Metrics Symposium (METRICS)*. 2005, 10–pp.

[30] Francis J. Lacoste. "Killing the Gatekeeper: Introducing a Continuous Integration System". In: *Agile Conference*. 2009, pp. 387–392.

[31] Wing Lam et al. "A Large-Scale Longitudinal Study of Flaky Tests". In: *Proceedings of the ACM on Programming Languages* (2020), pp. 202–231.

[32] Wing Lam et al. "A study on the lifecycle of flaky tests". In: *International Conference on Software Engineering (ICSE)*. 2020, pp. 1471–1482.

[33] Wing Lam et al. "Root Causing Flaky Tests in a Large-Scale Industrial Setting". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2019, pp. 204–215.

[34] Wing Lam et al. "Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects". In: *International Symposium on Software Reliability Engineering (ISSRE)*. 2020, pp. 403–413.

[35] Fabian Leinen et al. "Cost of Flaky Tests in Continuous Integration: An Industrial Case Study". In: *International Conference on Software Testing, Verification and Validation (ICST)* (2024).

[36] H.K.N. Leung and L. White. "Insights into regression testing (software testing)". In: 1989, pp. 60–69.

[37] Jeff Listfield. *Where do our flaky tests come from?* URL: `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html` (visited on 2024-05-21).

[38] Qingzhou Luo et al. "An empirical analysis of flaky tests". In: *International Symposium on Foundations of Software Engineering (FSE)*. 2014, pp. 643–653.

[39] John Micco. "The state of continuous integration testing@ google". In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2017.

[40] Jesús Morán et al. "Flakyloc: flakiness localization for reliable test suites in web applications". In: *Journal of Web Engineering* 19 (2020), pp. 267–296.

[41] Meiyappan Nagappan and Mladen A Vouk. "Abstracting log lines to log event types for mining software system logs". In: *International Conference on Mining Software Repositories (MSR)*. 2010, pp. 114–117.

[42] Owain Parry et al. "Surveying the developer experience of flaky tests". In: *International Conference on Software Engineering (ICSE)*. 2022, pp. 253–262.

[43] Yu Pei et al. "An empirical study of async wait flakiness in front-end testing." In: *BENEVOL*. 2022.

[44] Akond Rahman et al. "Characterizing the influence of continuous integration: empirical results from 250+ open source and proprietary projects". In: *ACM SIGSOFT International Workshop on Software Analytics*. 2018, pp. 8–14.

[45] Pilar Rodriguez et al. “Continuous deployment of software intensive products and services: A systematic mapping study”. In: *Journal of Systems and Software* (2017), pp. 263–291.

[46] Alan Romano et al. “An Empirical Analysis of UI-based Flaky Tests”. In: *International Conference on Software Engineering (ICSE)*. 2021, pp. 1585–1597.

[47] Felix Salfner and Steffen Tschirpke. “Error Log Processing for Accurate Failure Prediction.” In: *WASL* (2008), p. 4.

[48] Ray Schneider. “Continuous integration: improving software quality and reducing risk”. In: *Software Quality Professional* (2008), p. 51.

[49] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. “Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices”. In: *IEEE* (2017), pp. 3909–3943.

[50] August Shi, Peiyuan Zhao, and Darko Marinov. “Understanding and Improving Regression Test Selection in Continuous Integration”. In: *International Symposium on Software Reliability Engineering (ISSRE)*. 2019, pp. 228–238.

[51] August Shi et al. “iFixFlakies: A framework for automatically fixing order-dependent flaky tests”. In: *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019, pp. 545–555.

[52] Swapna Thorve, Chandani Sreshtha, and Na Meng. “An Empirical Study of Flaky Tests in Android Apps”. In: 2018, pp. 534–538.

[53] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. “An empirical study of bugs in test code”. In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 101–110.