Cost of Flaky Tests in Continuous Integration: An Industrial Case Study

Fabian Leinen
Technical University of Munich
Munich, Germany

Daniel Elsner
Technical University of Munich
Munich, Germany

Alexander Pretschner
Technical University of Munich
Munich, Germany

Andreas Stahlbauer
CQSE GmbH
Munich, Germany

Michael Sailer
CQSE GmbH
Munich, Germany

Elmar Jürgens
CQSE GmbH
Munich, Germany

Abstract—**Researchers and practitioners alike increasingly often perceive flaky tests as a major challenge in software engineering. They spend a lot of effort trying to detect, repair, and mitigate the negative effects of flaky tests. However, it is yet unclear where and to what extent the costs of flaky tests manifest in industrial Continuous Integration (CI) development processes.**

In this study, we compile cost factors introduced by flaky tests in CI development from research and practice and derive a cost model that allows gaining insight into the costs incurred. We then instantiate this model in a case study of a large, commercial software project with $\sim$30 developers and $\sim$1M SLoC. We analyze five years of development history, including CI test logs, commits from the Version Control System (VCS), issue tickets, and tracked work time to quantify the cost factors implied by flaky tests. We find that the time spent dealing with flaky tests in the studied project represents at least 2.5% of the productive developer time. This effort is divided into investigating potentially flaky test failures, which accounts for 1.1% of the total time spent, repairing flaky tests adds another 1.3%, and developing tools to monitor flaky tests adds 0.1%. Contrary to most other studies, we find the cost for rerunning tests to be negligible and inexpensive. Automatically rerunning a test costs 0.02 cents, while not rerunning and thus letting the pipeline fail results in a manual investigation costing $5.67 in our context. The insights gained from our case study have led to the decision to shift effort from investigation and repair to automatically rerunning tests.

Our cost model can help practitioners analyze the cost of flaky tests in their context and make informed decisions. Furthermore, our case study provides a first step to better understand the costs of flaky tests, which can lead researchers to industry-relevant problems.

Index Terms—**flaky tests, continuous integration, regression testing, cost modeling, industrial case study**

I. Introduction

Regression testing is a testing activity commonly performed in CI environments to ensure that introduced changes do not break existing system behavior [1, 2, 3]. The developers' trust in the correctness of the software depends on the reliability of test results [4]. A test case that shows different test results on two different occasions yet with identical environmental factors (for example, the same code under test) is commonly referred to as a *flaky test* [5]. According to recent developer surveys [6, 7, 8] and industrial studies [4, 5, 9], flaky tests are a growing problem in regression testing and a major source of development costs [10, 11]. Microsoft [4] and Google [10], for instance, report 4%–16% of tests to involve flakiness and 1.5% of CI test runs to be flaky. Therefore, researchers and practitioners invest a lot of effort to understand flaky tests [12, 13] and to mitigate their negative impacts by automatically detecting flaky tests [14, 15, 16], determining their root causes [4, 17, 18], and repairing them [19, 20].

The existing studies either provide a qualitative overview of the life cycle of flaky tests [12, 14, 21, 22, 23, 24, 25, 26, 27, 28] or technical solutions for one phase of the life cycle [15, 16, 17, 18, 29, 30, 31, 32, 33, 34, 35]. However, it is unclear how different phases contribute to the overall cost of flaky tests in CI development processes.

In this study, we analyze and quantify the factors driving the cost behind flaky tests in industry-scale CI development environments together with our industry partner CQSE. Therefore, we identify both well-studied and relatively unknown cost factors from research and practice, synthesize them into a cost model, and discuss common strategies for how these factors are balanced in different contexts. Then, we conduct a case study on up to five years of the development history of a large-scale software project at CQSE, where we quantify and further flesh out the cost factors from our model. In particular, we (1) revisit the costs for rerunning failed tests and thereby detecting flaky failures [10, 14, 36], and study the so far widely unknown cost incurred in (2) investigating test failures, (3) repairing flaky tests, and (4) managing and developing flaky test monitoring.

The results of our case study indicate that in the given context, the cost for automatically rerunning failed tests is negligible at $3 per month (0.63% of all test executions). In contrast, at least 2.5% of total developer time is spent investigating, repairing, or managing flaky tests, for example, monitoring them. We find these numbers are likely a lower bound by asking developers from CQSE for their feedback and experience with different cost factors related to flaky tests.

To be precise, a failed test results in a failed pipeline, which requires average costs of $5.67 worth of manual investigation by developers. In the case of a flaky test failure, rerunning the test can suppress the failure, at a cost of $0.02. Based on these findings in this specific context, we recommend shifting effort from expensive developer time to relatively cheap compute time by increasing the maximum number of reruns for failed test cases. These test cases are regression tests that support the decision as to whether the commit under test can be merged safely, that is, whether the commit will not introduce new bugs. Consequently, assuming that flaky failures are caused by a defect in the System under Test (SuT) that existed prior to the commit under test, ignoring them is a valid option. Nevertheless, as flaky failures might still indicate a problem in the SuT, the flaky behavior of tests needs to be monitored and, if necessary, repaired. At CQSE, this strategy is implemented by automatically rerunning failed tests up to five times instead of once, and logging flaky failures. To maintain a high-quality test suite and prevent non-deterministic defects from going into production, the most frequently failing tests are examined during weekly assessments and tickets are created to repair them. After implementing the new strategy for five months, we find that 4.8% of all test suite executions, or every $\sim 20^{\text{th}}$ test suite execution, has at least one flaky failure that is detected by the new strategy and would have gone unnoticed by the previous strategy.

In summary, this paper makes the following main contributions:

- **Flaky Test Cost Model:** Our study is the first to compile cost factors from research and practice to derive a cost model for flaky tests in CI.
- **Industrial Case Study:** We analyze 5 years of development data from an industry-scale CI development environment to understand costs related to flaky tests. At CQSE, our insight led to a more cost-efficient strategy for handling flaky tests.

II. Cost Model

As the cost of flaky tests in CI development processes is largely unstudied, we need to identify the most relevant cost factors associated with flaky tests in CI development. This section first compiles a set of cost factors and their context-specific input parameters from research and practice. These cost factors form a cost model, which aims to create transparency about costs incurred by flaky tests. On the one hand, this can help practitioners properly allocate resources for dealing with flaky tests along with domain knowledge. On the other hand, it can help researchers identify relevant problems for their research. The way we structure our cost model into cost factors and their context-specific input parameters is inspired by previous research on cost modeling in the context of regression testing [37, 38]. Based on this model, we then present common strategies for dealing with flaky tests from different practical contexts and discuss scenarios where one strategy outperforms another. While we do not expect our cost model to be universally complete in every context, it consolidates previous research and practical insights and thus serves as a guideline for cost analysis in CI development.

A. *Cost Factors*

Below we present cost factors derived from previous studies and discussions with our industry partner CQSE. Each cost factor depends on input parameters that are specific to a given context or project and aims to represent the cost for a predefined period (for example, for *one month*, to reason about *total monthly cost*).

1) Test Case Rerunning: The goal of regression testing is to detect regression bugs introduced by the commit under test, and thereby decide whether the commit can be merged safely [2]. Assuming that the reason for a flaky failure is usually not introduced by the commit under test, flaky failures can be ignored when deciding whether to merge a commit. However, they do demand attention during subsequent processes. One might argue that a test can also pass flakily, and thus require reruns. In CI regression testing, tests are often run against small changes, so most tests rarely cover changes. Thus, they run under approximately the same conditions many times in the normal development workflow, resulting in the identification of non-deterministic issues without explicitly rerunning potentially flaky passes. In addition, developers perceive consequences from flaky failures, such as false alerts, as more critical than undetected bugs resulting from "flaky passes" [26, 14]. We therefore consider this well-established industry practice of rerunning failed test cases to be a reasonable approach.

In academia, the supposedly high cost of rerunning failed tests is often used to motivate the need for new approaches to detect flaky failures or flaky tests [15, 34, 31, 39, 16, 40]. In line with the literature on rerunning flaky tests [31, 14, 15], the cost for test case running, C_{rerun}, is driven by the context-specific input parameters n_{rerun}, the number of test case reruns, and c_{test}, the cost for executing a test case.

$$C_{\text{rerun}} = n_{\text{rerun}} \cdot c_{\text{test}} \tag{1}$$

If tests tend to be particularly flaky, n_{rerun} will be higher. While c_{test} is typically driven primarily by test duration and the cost for compute time, there may also be significant test or infrastructure setup costs associated with rerunning tests, for example, in the context of hardware-in-the-loop test stands [41].

2) Failure Investigation: CI pipeline failures can indicate newly introduced regressions caused by changes to the software. Accordingly, developers must investigate each pipeline failure to determine the cause [42].

The effort of investigating pipeline failures is measured in the time spent by developers t_{inv}. The cost for pipeline investigation C_{inv} is calculated by multiplying the time spent t_{inv} by the hourly developer rate c_{dev}. As we will discuss in Section IV-B, restarting a job that failed due to a test is a common strategy to verify that a test failed flakily. Therefore, we also include the number of pipeline jobs restarted, n_{restart}, and multiply it by the cost per job run, c_{job}.

$$C_{\text{inv}} = t_{\text{inv}} \cdot c_{\text{dev}} + n_{\text{restart}} \cdot c_{\text{job}} \tag{2}$$

3) Repairing Flaky Tests: A flaky test is the observed non-deterministic behavior of a test case. Repairing a flaky test refers to the process of making a test case deterministic, or at least reducing the probability of an unintended verdict. Even though the term flaky test suggests that the reason for the non-deterministic behavior lies in the test, it can also lie in the SuT [21, 26]. Consequently, repairs of flaky tests can modify the test, the SuT, or both.

Repairing flaky tests can be tedious and time-consuming, yet crucial if the reliability of test results is to be increased [19]. While achieving complete stability may be unfeasible for certain test types, like User Interface (UI) tests [5, 9, 43], repairing such tests can still reduce their flakiness.

Similar to the cost for investigating pipeline failures, the cost for repairing flaky tests, C_{repair}, can be inferred from the cost for developers, c_{dev}, multiplied by the time to repair, t_{repair}.

$$C_{\text{repair}} = t_{\text{repair}} \cdot c_{\text{dev}} \tag{3}$$

Repairing flaky tests can include improvements to the SuT [12], which also incurs costs but also has a beneficial effect on the SuT.

4) Development Delays: Failing pipelines can delay merging and integrating changes [8] and ultimately negatively impact release plans [44]. In particular, if the round-trip time between branching and merging a change is high, the probability of merge conflicts increases, which can be costly to fix. Flaky failures increase this time, leading to higher merge and integration efforts and thus higher costs. On an organizational level, delaying releases or failing to ship a feature with a scheduled release due to blocked pipelines can reduce customer satisfaction, resulting in additional delay costs.

We denote the additional required development time caused by delays, as $t_{\text{delay dev}}$. Thus, the additional cost for delays is obtained by multiplying the additional development time $t_{\text{delay dev}}$ by the developer cost per time c_{dev} and adding the organization-wide delay costs $c_{\text{delay org}}$ (for example, reduced customer satisfaction or market competitiveness). Note that the organizational delay cost $c_{\text{delay org}}$ is already an amount of currency and is thus not multiplied by some cost per time. We decided to conclude it in one variable as it is not proportional to the delay time. A feature that is delayed but still released within the same release cycle will not affect customer satisfaction or market competitiveness and thus cost while only a slightly longer delay might if it causes the feature to be released in the next release cycle.

$$C_{\text{delay}} = t_{\text{delay dev}} \cdot c_{\text{dev}} + c_{\text{delay org}} \tag{4}$$

5) Flaky Test Management: Several studies [21, 45] and blog articles (for example, from Spotify [46] and GitHub [47]) describe efforts in building flaky test management systems to store, visualize, and manage information about flaky tests.

The costs for developing and using flaky test management systems, C_{manage}, are thus driven by the developer cost per time, c_{dev}, as well as the time spent for flaky test management efforts, t_{manage}:

$$C_{\text{manage}} = t_{\text{manage}} \cdot c_{\text{dev}} \tag{5}$$

Note that this cost factor may also include additional developer efforts related to systematically analyzing a system (for example, using static or dynamic program analysis [4]) to better understand sources of flakiness as a whole, besides investigating or repairing specific flaky tests.

6) Production Bugs: Rahman and Rigby show that ignored (or *quarantined* [49]) flaky tests can degrade the product quality [48], which can, for instance, manifest in production bugs encountered by customers.

We define n_{bug} to be the number of missed bugs that go to production and c_{bug} the average cost per production bug.

$$C_{\text{bugs}} = n_{\text{bug}} \cdot c_{\text{bug}} \tag{6}$$

The cost per production bug can be influenced by directly incurred costs, such as liability issues, or indirect costs, such as loss of prestige and, thereby new customers. These costs also include the cost for fixing the bug, which is expected to be generally higher after release, as localizing and fixing the cause may take longer in hindsight [38, 50].

B. Total Cost

The cost factors form a cost model, which can be used to calculate the total costs induced by flaky tests. The total cost of flaky tests in CI, C_{total}, is calculated as the sum of the individual cost factors in a given context:

$$C_{\text{total}} = C_{\text{rerun}} + C_{\text{inv}} + C_{\text{repair}} + C_{\text{delay}} + C_{\text{manage}} + C_{\text{bugs}} \tag{7}$$

Note that this model aims to evaluate the cost of flaky tests in a static environment, not to predict the impact of future decisions. However, the insights it provides and expert knowledge about a specific project can help guide decisions.

C. Cost for Strategies in Practice

Existing studies of flaky tests present different strategies for dealing with flaky tests in different contexts. To demonstrate how the cost model can be used to compare such strategies, we briefly discuss four simplified strategies:

- *Heavy-rerun*: The *Heavy-rerun* strategy implies that tests are rerun many times before a failure is reported as such. By not reporting flaky failures but ideally only deterministic ones, the cost for investigating flaky failures is reduced. The cost of this strategy is mainly driven by C_{rerun} and if the flakiness originates from the SuT, C_{bugs}. A variant of *Heavy-rerun* has reportedly been used in large-scale industrial software at Google [10] and SAP [40].
- *Immediate-fix*: With the *Immediate-fix* strategy, developers are encouraged to investigate and repair flaky tests when they first occur. The costs are mainly C_{inv} and C_{repair}. We have found one mention of this strategy in the regression testing literature [51] and a moderate form at Microsoft [21].
- *Quarantine-test*: The *Quarantine-test* strategy suggests quarantining, that is, ignoring known flaky tests to prevent them from slowing down the development process. Although initially seeming plausible, a case study has shown that flaky tests have a strong ability to reveal deterministic faults [52]. When quarantining, the main cost factor is

C_{bugs}, as production bugs may occur more frequently if critical parts of the software are not covered by tests other than the ignored ones [28, 48, 49, 52]. Reportedly, the *Quarantine-test* strategy has also been used at Google by quarantining flaky tests [49, 53].

- *Investigation-only*: The *Investigation-only* strategy basically implies that there is no systematic way to handle flaky tests. Developers are responsible for individually investigating the cause of pipeline failures and deciding whether the change can still be integrated. The driving cost factors are thus C_{inv} and C_{bugs} since developers may fail to assess the situation correctly. We expect such a strategy to be useful for small or rather early-stage projects, where flaky tests are rare and a pragmatic ad hoc strategy makes sense.

III. Study Setup

We will instantiate the model derived in Section II at our industry partner CQSE in Section IV. Therefore, in this section, we describe the context and the data sources we use to quantify the cost factors in detail.

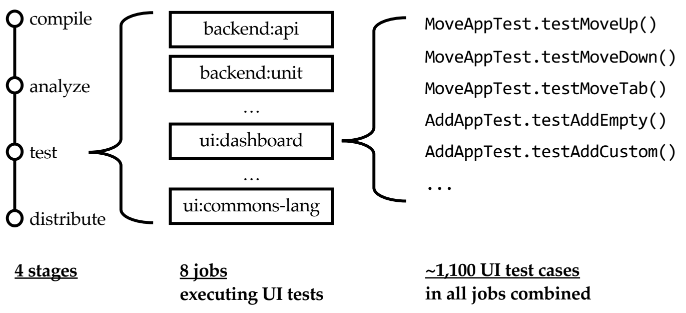
A. Study Subject

CQSE operates with a widely adopted tech stack and adheres to common development practices. To facilitate a deeper comprehension of our case study, we describe both, the technical and the process-related aspects with a particular emphasis on flaky tests.

1) System Description: The software system analyzed in this study (see our case study in Section IV) has ∼1M Source Lines of Code (SLoC) stored in a monolithic code repository and ∼30 developers actively developing on it. The software is a distributed system. The backend is primarily developed in Java, runs multiple jobs for computationally intensive analysis tasks, runs user management functionality, and provides a REST API for the front-end. The front end is developed in Typescript and provides a web-based user interface for the analysis results and allows users to configure the analysis jobs.

2) Development Process: Throughout the software development process, JIRA is used to track all issues, such as new features, bugs, or flaky tests that need to be repaired. Tickets are assigned to developers, who typically create a new `git` branch for each ticket and develop on it locally. These feature branches are usually directly merged into the *main* branch. Release branches, branching of the main branch, are created for every major or minor release version increment, for example, *2.8.x*, *2.9.x*, and *3.0.x*.

Each code submission to the code repository (that is `git push`) triggers a GITLAB CI *pipeline*, which is depicted in its rough structure and dimensions in Figure 1. A pipeline consists of four *stages* for compiling, analyzing, testing, and distributing the code (packaging and deploying). The testing stage runs the regression test suite in 16 *jobs*, 8 of which run UI tests. The UI tests are particularly interesting in our study because they are most prone to flakiness. Each UI test job executes, on average, about 150 test cases, making a total of ∼1,100 UI test cases in a single pipeline. Currently, a total of ∼2,000


compile
analyze
test
distribute
backend:api
backend:unit
…
ui:dashboard
…
ui:commons-lang
MoveAppTest.testMoveUp()
MoveAppTest.testMoveDown()
MoveAppTest.testMoveTab()
AddAppTest.testAddEmpty()
AddAppTest.testAddCustom()
…
4 stages
8 jobs executing UI tests
~1,100 UI test cases in all jobs combined
Fig. 1: Structure and dimensions of the CI pipeline.

pipelines are executed per month. In the case of a passing pipeline, the development continues normally. On the other hand, a failed pipeline needs to be manually investigated, and can have different reasons why the pipeline failed (we discuss these in more detail in Section IV-B), such as deterministic test failures. Developers can fix these according to their typical workflow. In contrast, if the developer suspects the test failure to be a *flaky* failure, the process is as described next.

3) Flaky Test Process: Firstly, to ensure that the failure was not deterministic but flaky, the developer can manually restart the entire pipeline job. Secondly, if the developer is certain that the failing test cannot exercise the changed code and the test is known to be prone to flakiness, the failure can be ignored. Thirdly, in consultation with the reviewer, a change may even be merged despite a failed test. In any case, if the test is newly identified as flaky, a ticket is created in JIRA to repair that test.

At CQSE, flaky failures most often occur in automated UI tests. With these tests, no functionality is mocked, so they test the system end-to-end. All UI tests are considered potentially flaky and are rerun exactly once after failure. For tests that frequently fail flakily, the maximum number of reruns can be temporarily set to 10, but in this case, a high-priority JIRA ticket will be created to repair that test in the short term. For tests other than UI tests, no automatic rerunning is applied. Flaky failures detected by rerunning are centrally collected in an internal tool called *Flickering Tracker*. This tool, developed since 2019, allows for tracking the flakiness of tests. A timeline of each test shows when it failed flakily, and information such as the stack trace and possibly screenshots are provided for each flickering failure. Through a JIRA integration, existing tickets for each test are displayed, and a new ticket can be created to fix it. As of today, the Flickering Tracker is being used and developed by a pipeline stability project group.

This effort shows, among other things, that the awareness of flaky tests is deeply rooted among engineers, and accordingly, great efforts are already being made to mitigate their negative effects. Besides rerunning and monitoring, no automated techniques to mitigate the negative effects of flaky tests are applied to date, as it is common in practice [8]. Also, no "investigation aid" like a flaky failure detector [15, 31], or automatic approach to find the location [29] or root cause [4, 17] is applied.

B. Data Sources

To quantify the cost factors we need to determine the input parameters needed to calculate the individual cost factors. To do this, we access data from the company's JIRA, GIT, GITLAB, and SLACK systems. Below, we describe the available data points and how we use them in our case study. Table I provides an overview of the data sources and the period we use for our case study.

TABLE I: Data sources used in our case study.

System	Data	Period
Jira	Booked time for investigating pipeline failures	6 months
	Tickets incl. description, booked work time, ...	5 years
Git	Changes made in every JIRA ticket	5 years
Gitlab	CI data, mainly test executions	6 months
Slack	Manually labeled pipeline failure cause	6 months

Jira: At CQSE, the issue tracker JIRA contains all tickets from software development. Each ticket includes a description, comments, and the total work time booked. In addition, the corresponding GITLAB merge requests are automatically linked to the completed tickets.

Our study uses data from five years, from the beginning of 2018 to the end of 2022. As cost is the central aspect of our study, we use the booked work time per ticket. To put the absolute numbers into perspective, we additionally report the total time spent on product development by summing up the times booked on all tickets in the investigated period.

Git: For each JIRA ticket that is resolved, we use the corresponding Git commit to extract the changes. This is used to determine if flaky tests have been repaired in the SuT or the test (see Section IV-C). This information is available for the same period as the JIRA data.

GitLab: Every resolved JIRA ticket is connected to a merge request in GITLAB. For each merge commit in the closed merge requests, a pipeline is executed. A pipeline is also triggered for each push to GITLAB. We query all pipelines and their jobs executing UI tests with available metadata, for example, job results and duration. We parse the jobs' logs to extract individual test execution verdicts and rerun counts.

This data is used for the six months from September 2022 to February 2023. Due to a recent migration of the GITLAB system, no data is available before this period. For this timespan, we collect 13,300 pipelines with 90,500 jobs executing UI tests, including 10.3M individual test runs. Of these pipelines, 6,880 failed, and of those 2,720 failed due to failing UI tests.

Slack: A failing pipeline targeting the main or a release branch triggers a bot to post this event in a dedicated developer channel in SLACK. These events are investigated by the pipeline officer, a role assigned weekly to a member of the pipeline stability project group. The pipeline officer labels the pipeline failure according to the reason for failure in SLACK with one of four categories: *build failure* (for example, compilation errors due to integration issues), *deterministic test failure*, *flaky test failure*, and *infrastructure problem* (for example, connection to external services such as a remote artifact repository). Additionally, the pipeline officer takes some action due to the failure and adds a label for the action taken in SLACK. This could be one of *quick fix* (the issue was fixed with the next commit), *restart* (the job was manually restarted), or JIRA (a ticket was created for the failing test). Since this process is relatively new, we also use the data from September 2022 to February 2023.

IV. CASE STUDY

To obtain transparency and a detailed understanding of costs for flaky tests in CI for the previously described context, we instantiate the cost model presented in Section II in a real-world case study at CQSE. Consequently, in the first step, we must identify the relevant cost factors in this context. We do this in collaboration with developers from CQSE and find the factors *test case rerunning* (C_{rerun}), *failure investigation* (C_{inv}), *flaky test repair* (C_{repair}), and *flaky test management* (C_{manage}) to be most relevant for the software project at hand. Contrary to our initial assumptions, costs for potentially missed *production bugs* (C_{bugs}) and *development delays* (C_{delay}) are not perceived as a relevant problem as reported by developers. Moreover, developers perceive flaky tests as a problem mostly with UI tests (see Section III-A), so we limit the cost analysis in our case study to these types of tests.

After identifying the relevant cost factors in this context, we formulated the following four Research Questions (RQs) to quantify each factor and steer the case study:

RQ1 What is the cost for *automatically rerunning* failed tests?
RQ2 What is the cost for *investigating* failed pipelines?
RQ3 What is the cost for *repairing* flaky tests?
RQ4 What is the cost for *management* of flaky tests?

We further defined costs for *one month* as the target granularity for estimating the total costs. Therefore, the cost factors quantified based on the answers to all RQs are averaged to fit this granularity.

A. Cost for Automatically Rerunning Tests

1) Approach: A log capturing the standard output and error streams is available for each job. By parsing all logs from September 2022 to February 2023, we obtain 10,266,054 individual test executions (including reruns). These test executions are split over 1,166 different test cases. For every test case, between 1 and 17,517 executions, with a mean of 8,812 and a median of 10,970, are obtained. Of these test cases, 74 have 27 or fewer executions, while the others have at least 103 executions. These test cases with few executions are either newly introduced or have been added for a short period for debugging purposes. We exclude those with 27 or fewer executions from our analysis since they do not provide enough data to draw meaningful conclusions.

To estimate the influence of the maximum number of reruns on the number of flaky failing pipelines, we need to understand how flaky every test is. Henderson et al. denote the observed flake rate of a test case t, and hence its probability to fail flakily, as $\hat{f}_t = \frac{\text{failing runs}}{\text{total runs}}$ [54]. To obtain $\hat{f}_t$, we use the parsed logs. Of all 1,092 test cases with at least 103 executions, 357 flaked at least once, while 735 did not. Taking into account only test cases that flaked at least once, we find that the mean flake rate is 0.30%, the median 0.02%, the lower quartile 0.01%, the upper quartile 0.91%, and the maximum 10.44% over the entire six months.

To make further statements about hypothetical scenarios in addition to actual observations, we make the simplifying assumption that test runs would be independent and identically distributed, as implied by Kowalczyk et al. [9] and modeled every test run as a Bernoulli trial with a probability of failure of $\hat{f}_t$. Thus, running a test case multiple times can be modeled as a Binomial distribution $\mathcal{B}(n_{\text{max}}, \hat{f}_t)$ with n_{max} being the maximum number of reruns. The probability of at least one test failure in a pipeline, despite the automatic rerunning mechanism, and hence a pipeline failure is then given by

$$p_{\text{pipeline failure}}(n_{\text{max}}) = 1 - \prod_{t \in \text{tests}} (1 - \hat{f}_t^{1+n_{\text{max}}}) \qquad (8)$$

2) Results: We find that 99.37% of all test executions are no automatic reruns, which would have been executed just as well without the automatic rerun mechanism. The remaining 0.63% (66,375) are reruns triggered by the automatic rerun mechanism. This value breaks down into 0.09 percentage points of flaky failures and 0.54 percentage points of true test failures (tests that failed all reruns). The total run time of UI test jobs in the period examined was 55,073 h. The presence of flaky tests causes the need for the automatic rerun mechanism, so we attribute the cost for all reruns, even those caused by fault-revealing, to flaky tests. We, therefore, estimate the run time caused by flaky tests to be 0.63% of the total run time, or 347 h or 14.46 days.

Using Equation 8, we estimate the share of pipelines that would have failed despite $n_{\text{max}} = \{0, 1, .., 5\}$ maximum test reruns of the automatic rerun mechanism. Note that $n_{\text{max}} = 0$ reruns means that the initial run is still performed. We find that the probability of a pipeline failing flakily is 66.63% with no reruns, 3.41% with one rerun, 0.21% with two reruns, and 0.02% with three reruns. 0.00% is reached with four reruns. To validate the model, we compare the theoretical pipeline flaky failure probability with zero reruns (66.63%) with the actual one assuming there would be no automatic rerun mechanism. We find that in our study subject, 51.88% of pipelines had at least one test failure that was detected by the automatic rerun mechanism, which prevented the pipeline from failing. The pipeline flaky failure probability would have been 51.88% with zero reruns. We speculate that the difference between the actual pipeline flaky failure probability (51.88%) and the calculated one (66.63%) is due to our simplified assumption that test runs are independent and identically distributed, which is not entirely true in practice [15]. However, we believe this difference is slight, and the general model allows rough estimations. As briefly discussed in Section IV-C, the origin of flaky tests can be a non-deterministic occurring fault in the SuT and thus rerunning can mask newly introduced faults in the SuT.

3) Estimation of Cost Factor: As stated above, the total number of reruns for September 2022 to February 2023 is 66,375. This results in a monthly average of 11,063 reruns. The average run time per test case is 19.31 s. Our industry partner's computing power[1] is billed with $0.044\frac{\$}{h}$ resulting in $c_{\text{test}} = 19.31\text{s} \cdot 0.13\frac{\$}{h} = \$236 \cdot 10^{-6}$ (or 0.02 cents per automatic test rerun). We calculate the resulting cost C_{rerun} by inserting n_{rerun} and c_{test} into Equation 1:

$$\begin{aligned} C_{\text{rerun}} &= n_{\text{rerun}} \cdot c_{\text{test}} \\ &= 11{,}063 \cdot \$236 \cdot 10^{-6} \\ &= \$3 \end{aligned}$$

RQ1 (Automatic Rerunning): The monthly cost for automatically rerunning is \$3 caused by 0.63% of all test executions that were reruns. A single automatic rerun costs 0.02 cents. Automated reruns have shown to be an effective way to reduce the number of flaky pipeline failures, but vouches for the risk that faults in the SuT will go undetected.

B. Cost for Investigating Failed Pipelines

1) Approach: At CQSE, the investigation of failed pipelines usually starts in GitLab. Build failures and infrastructure problems can be quickly identified by the stage of the failing job or in combination with the error log. To differentiate between a flaky test failure and a deterministic test failure in a failing UI test job, developers usually compare the code section likely covered by the failing test (as there is no testwise end-to-end code coverage available for UI tests) with the changed code section. When the sections do not overlap, the failure is typically identified as flaky. In cases of overlapping sections, the investigation becomes considerably more complex, often necessitating local reproduction of the issue, which is both time-consuming and costly. In doubt, developers often restart an entire job to see if the test case fails again. Given that only that one test case is of particular interest, this imposes a significant overhead. We use the metadata of the executed jobs to measure how often these job restarts occurred and sum up the execution time.

A total of 6,878 pipelines failed during the analyzed period (2,718 of which failed due to failing UI tests). Of all failed pipelines targeting the main or release branch, 686 have a label indicating the cause of the failure. Most of these pipelines are assigned exactly one label; for 34 pipelines, multiple labels are assigned. For pipelines targeting changes of other branches, these labels are not available. Furthermore, the action taken in response to the failure is indicated in 157 cases. This action can entail job restarts, the creation of Jira tickets, or the designation of the failure as resolved with the next commit.

2) Results: By examining the logged work time in Jira, we find that the time spent on pipeline investigation during the studied period amounts to 25:52 h for labeling 686 pipelines. Consequently, the average time spent investigating a pipeline failure stands at 2:16 min. This analysis includes the time spent

[1]Spot instances of Google Cloud's e2-standard-4 machine

on pipeline failures stemming from all causes. However, for the purposes of this study, only the time spent due to flaky tests is relevant.

Taking into account all branches, a total of 6,878 pipelines failed between September 2022 and February 2023, with 2,718 of these failures attributed to at least one failing UI test. For other branches, such as feature branches, the time spent investigating pipeline failures is not separately tracked. We deem it reasonable to generalize the average time recorded for all types of failures to these other data. As previously mentioned, the identification of build failures and infrastructure problems is straightforward, while distinguishing between deterministic and flaky test failures is a labor-intensive process, as affirmed by developers. Consequently, we adopt the average time of 2:16 min per investigated pipeline failure as the lower limit for investigating pipeline failures caused by failing UI tests. To determine the time spent on all pipeline failures, not only the labeled set of failures, we multiply 2,718, the count of pipelines that failed due to UI tests, by the average time spent investigating pipeline failures (2:16 min), resulting in a total time spent of 102 h. This accounts for 1.11% of the total development time expended over these six months.

TABLE II: Labels for pipeline failures of 686 pipelines (36 of which have more than one label assigned).

Cause	Occurrences
Infrastructure problems	278
Flaky failures	206
of which UI tests	*96*
of which other tests	*110*
Build failures	134
Deterministic test failures	83
of which UI tests	*51*
of which other tests	*32*
Unknown	24

Table II provides an overview of the occurrence of all labeled causes for pipeline failures. The data reveals that infrastructure problems were the most prevalent cause for failed pipelines. These are usually problems with connecting to the repository manager or other external services that do not affect the testing stage. In total, 289 pipelines failed due to test failures, with flaky tests accounting for more than two-thirds of these instances (206). Furthermore, the data presented in Table II indicates that 96 pipeline failures were caused by flaky tests within the UI tests and 110 within other tests. Recall that automatic rerunning is only applied to UI tests. This highlights the significance of flaky tests in the context of UI tests, as more than half of the 13,300 pipelines would have failed due to flaky UI tests without the automatic rerun mechanism (see IV-A1). Also, there are nearly twice as many pipeline failures caused by UI test failures as there are true regression bugs.

Data about the action taken is available for 48 pipelines that failed due to flaky tests. The most common action is to restart the entire job, which occurred in 24 cases. In 22 instances, a JIRA ticket was created, and in two instances, the issue was immediately resolved with the next commit.

From these data, it is evident that manually restarting jobs is a common strategy for investigation. During the analyzed period, 2.05% (1,858) of all UI test jobs were manually restarted, accounting for 2.81% (1,493 h) of the total runtime for UI test jobs. The specific reasons behind each job restart are not documented, but developers report that this was exclusively done to identify flaky failures.

In discussions with developers, they confirm that the average time for investigating a pipeline failure of 2:16 min seems plausible or rather too low. This time does not include context switches, which have been shown to significantly negatively impact developer productivity [55]. In modern development cycles, where fast delivery plays a crucial role, large regression test suites and a high number of pipeline executions are indispensable. Given the substantial number of pipeline runs and consequently the occurrence of failing pipelines, the time to investigate test failures within CI pipelines holds significance.

3) Estimation of Cost Factor: Over six months under review, the time spent on investigating pipeline failures amounts to 102 h. This equates to an average of $t_{\text{inv}} = 17$ h per month. Additionally, we found that 1,858 jobs were manually restarted, yielding a monthly average of $n_{\text{restart}} = 310$. As we cannot disclose our industry partner's hourly rate for developers, we set c_{dev} to 150\$/h for our calculations. Thus, we find the developer cost for investigating one pipeline failure to be \$5.67. The mean runtime of a UI test job is 36:30 min, calculated by dividing the total runtime over two months (55,073 h) by the total count of UI test jobs (90,500). Multiplying this average job time by the hourly rate of computational power at \$0.044 yields $c_{\text{job}} = 0.027\$$. Utilizing these values and Equation 2, the total cost for investigation C_{inv} for one month can be calculated:

$$\begin{aligned} C_{\text{inv}} &= t_{\text{inv}} \cdot c_{\text{dev}} + n_{\text{restart}} \cdot c_{\text{job}} \\ &= 17\text{h} \cdot 150\$/\text{h} + 310 \cdot \$0.027 \\ &= \$2{,}550 + \$8 \\ &= \$2{,}558 \end{aligned}$$

RQ2 (Investigating): Examining a single pipeline failure requires, on average 2:16 min causing costs of \$5.67. Due to the large number of pipeline runs and the resulting large number of pipeline failures, the time spent investigating pipeline failures is 1.11% of the total developer time. The monthly cost incurred is \$2,558.

C. Cost for Repairing Flaky Tests

1) Approach: Using JIRA, we search for tickets that either have "flicker" or "flaky" in the title and were resolved between January 1, 2018, and December 31, 2022, resulting in 416 tickets. We manually filter out 66 false positives, among others, tickets related to the development of the *Flickering Tracker*. Other excluded tickets described bugs with "flickering" animations or flaky tests that are not UI tests. After filtering, 350 tickets remain, of which 250 are marked as *Done* and 100 as *Discarded*. The primary reason for discarding is duplicate

TABLE III: Time spent on repairing flaky tests compared to the total time invested in product development.

	Repairing [h]	**Total [h]**	**Rel. [%]**
2018	133	14,600	0.91
2019	128	15,500	0.86
2020	213	20,200	1.10
2021	192	17,500	1.28
2022	211	18,300	1.15
Total	**877**	**84,700**	**1.04**

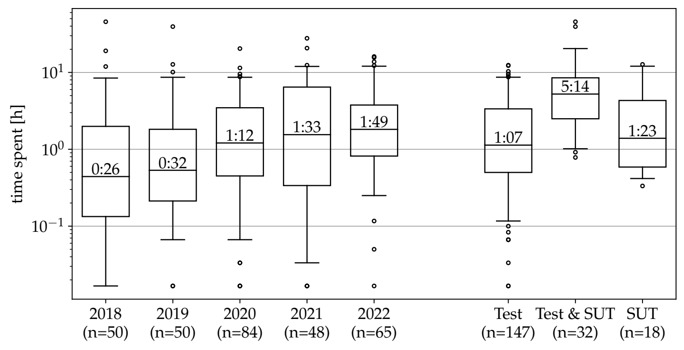


Fig. 2: Time spent per ticket grouped by year and region repaired

tickets. In some cases, tickets are discarded because the flaky behavior could not be reproduced. Anyhow, until a decision is made to reject a ticket, the time has already been spent on the respective ticket, which has also been booked. Booked times are available for 297 tickets (241 marked as *Done*, 56 as *Discarded*).

For 241 completed tickets, a merge request is linked, and the time spent on the ticket is available. We use GITLAB's REST API to gather information about the merge request, particularly the source commit hash and target commit hash. Using `git`, we extract the changes made in each of the tickets on a file level. Due to unavailable commits, for example, due to rebasing and housekeeping, this information cannot be obtained for five tickets, leaving 236 tickets with this information available.

Based on the file path and name, we classify each change into the part it changed, either the SuT or the test itself. Thereby, we filter out irrelevant changes, such as formatting adjustments or changing the annotation from one that reruns a failed test up to 10 times to the standard one that only reruns it once. Out of all the merge requests, 39 only change the test annotation; that is, close the ticket without making any relevant changes. We filter out these tickets, leaving 197 tickets with relevant changes. Each ticket exclusively changes either the SuT or the test, or in some cases, both parts are changed. We determine this information by aggregating all changes for a merge request associated with a ticket.

2) Results: Using the time booked repairing flakiness across all tickets, we find that 877 h were booked during the five-year study period. The total time spent on product development within this period accounts for 84,700 h. Throughout the studied period, 1.04% of developer hours were spent on repairing flaky tests. Table III shows the time spent on repairing flaky tests broken down by year. Notably, the percentage of developer time spent on repairs increases over the years, peaking at 1.28% in 2021 before decreasing slightly in 2023. The left part of Figure 2 shows that the time spent per ticket constantly increases. In 2018, the median time booked for each ticket was 27 min, reaching its maximum of 109 min in 2022. This suggests that as system complexity increases and test suites run longer, reproducibility and repair become more difficult.

As changes to the SuT not only cause costs but might also be beneficial for the developed product, we investigate changes made when resolving these tickets. We find that 75% of all repairs exclusively change the test, not the SuT. Of all repairs, 16% involve changes to both the SuT and the test, while 9% exclusively change the SuT. The right part of Figure 2 breaks down the time booked into these three categories. For repairs that exclusively change the test, the median time per ticket is 1:08 h, amounting to a total of 346 h spent on such repairs. In the case of repairs changing both the test and the SuT, the median time per ticket is 5:14 h, with a total time of 260 h. Repairs exclusively modifying the SuT exhibit a median of 1:24 h per ticket, and a combined total of 58 h was expended on these repairs. In total, almost half of the time was spent on tickets that at least also change the SuT and thus have potentially added value for the product.

3) Estimation of Cost Factor: To calculate the cost for repairs C_{repair} over a one-month period, we use the time booked for this task t_{repair} and the developers' hourly rate c_{dev}. For the calculation of average monthly time spent, we use 877 h (from Table III) and a period of five years or 60 months. Thus, we set $t_{\text{repair}} = 15\text{h}$ for the target granularity of one month.

$$\begin{aligned} C_{\text{repair}} &= t_{\text{repair}} \cdot c_{\text{dev}} \\ &= 15\text{h} \cdot 150\$/\text{h} \\ &= \$2{,}250 \end{aligned}$$

RQ3 (Repairing): In the investigated project, up to 1.28% of the developer's time was spent repairing flaky tests. The monthly cost spent on repairing is $2,250.

D. Cost for Managing Flaky Tests

1) Approach: In our specific context at CQSE, we are aware of various efforts related to flaky test management. These include the development and maintenance of the internal tool *Flickering Tracker*, the development of a custom rerun mechanism, efforts for annotating tests and creating tickets, and improving the overall build stability, for example, by improving the infrastructure. While only the efforts associated with developing the Flickering Tracker have been documented in separate tickets, we limit ourselves to this activity. Similar to the approach outlined in Section IV-C, we queried JIRA for tickets with *Flickering Tracker* in the ticket title. By doing so, we received 16 tickets between April 2020 and December 2022 and manually verified that all of them were related to the development of the Flickering Tracker. However, it is important to mention that these tickets only cover the developmental history of the Flickering Tracker after its initial development and integration into the pipeline. As the development of this tool is an ongoing process, numerous open feature requests without booked times are excluded from this analysis.

2) Results: The total time spent on the development of the Flickering Tracker during the study period was 78 h. This represents 0.14% of developers' total development time for 2020, 2021, and 2022 (see Table III). We are aware that this number by no means includes all costs. Nevertheless, it can be argued that this effort probably makes good business sense, as visualization tools were reported as the top desire for information and tools in a developer survey [14].

3) Estimation of Cost Factor: The calculation for the monthly cost for managing flaky tests C_{manage} follows Equation 5. As the tracked 78 h discussed in Section IV-C were booked across 33 months, we calculate the average time for one month and set t_{manage} to 2.4 h. Again, the hourly rate for developers is assumed to be 150 \$/h.

$$\begin{aligned} C_{\text{manage}} &= t_{\text{manage}} \cdot c_{\text{dev}} \\ &= 2.4\text{h} \cdot 150\$/\text{h} \\ &= \$360 \end{aligned}$$

RQ4 (Management): The effort we were able to include in our calculation represents at least 0.14% of the developer time. The corresponding monthly costs would thereby be \$360.

E. *Estimation of Total Costs*

Following Equation 7, the total cost C_{total} can be calculated as the sum of the individual factors. As previously discussed, we received feedback from CQSE developers that C_{delay} and C_{bugs} are negligibly small and therefore set to 0 in this context. We have calculated C_{rerun}, C_{inv}, C_{repair}, and C_{bugs} for one month in RQ1-4 and insert them into Equation 7 to calculate C_{total}:

$$\begin{aligned} C_{\text{total}} &= C_{\text{rerun}} + C_{\text{inv}} + C_{\text{repair}} + C_{\text{delay}} + C_{\text{manage}} + C_{\text{bugs}} \\ &= \$3 + \$2{,}558 + \$2{,}250 + 0 + \$360 + 0 \\ &= \$5{,}171 \end{aligned}$$

Thus, in this context, the lower bound for the monthly cost for flaky tests is \$5,171.

V. Practical Implications

Referring back to the strategies detailed in Section II-C, we conclude that CQSE mostly follows a blend of the *heavy-rerun* and *investigation-only* strategies. However, no strategy is implemented to its full, extreme form. Within our context, a pipeline failure amounts to a manual investigation cost of \$5.67, while automatically rerunning a failed test case, potentially preventing a pipeline failure, incurs a mere cost of 0.02 cents.

This suggests that effort should be shifted from manually investigating pipeline failures to automatically rerunning test cases, so the *heavy-rerun* strategy should be pursued more aggressively. As flaky failures are typically unrelated to the tested changes, they can be ignored when deciding whether or not to merge a commit, possibly reducing investigatory costs, C_{inv}, due to fewer pipeline failures. Since flaky failures are still an important signal that may indicate a non-deterministically occurring defect in the SuT (which increases C_{bugs}), test cases that fail flaky the most should be investigated and repaired.

As a response to these conclusions, CQSE has implemented a new strategy for dealing with flaky test failures, favoring detection of flaky failures through test case reruns – up to five times instead of just once – with the most failure-prone test cases manually investigated and possibly repaired weekly. Implemented since March 15, 2023, this strategy significantly reduces C_{inv} as it suggests weekly, instead of per failure, investigations.

To evaluate the effectiveness of the new strategy, we examine the period of five months post-implementation. Test results are obtained in the same manner detailed in Section IV-A1. Of the 9417 pipelines and 10.6M test executions (including reruns) during these five months, 3631 pipelines had at least one flaky test failure. Nearly 3182 of these would have passed with only one automatic rerun as well. However, 449 pipelines necessitated more than one rerun. This constitutes 4.8% of all pipelines. No more undetected bugs were identified by the developers, nor were there any perceptible changes in addressing flaky test repairs.

VI. Threats to Validity

External Threats: The static nature of our cost model limits its applicability as a decision-making tool. While this may limit the generalizability of the model, it still provides valuable insight and knowledge about a particular project. Modeling the impact of decisions, such as increasing repair time, requires additional project-specific assumptions that are beyond the scope of this study. Additionally, Our cost model may omit cost factors or context-specific parameters significant in various contexts. To mitigate this, we base our assumptions on existing flaky testing literature and feedback from CQSE engineers with long-standing experience in multiple industrial contexts. Furthermore, as with most industrial case studies, we cannot and do not claim that our results generalize to other projects within CQSE, different companies, tech stacks, or timeframes. Despite this, we intentionally designed our cost model to be generic, anticipating other contexts will exhibit various trade-offs between cost factors.

a) Internal Threats: An internal threat stems from potentially inaccurate or incomplete development data, which we use in our case study at CQSE. Booked work time might be inaccurate or the costs included for the *Flickering Tracker* might not fully cover flaky test management costs. While this underestimates the actual cost, we addressed this threat by regularly discussing our case study's methodology and results with CQSE engineers for validation. Another threat arises from our evaluation scripts for data analysis and automatic data extraction from Jira, GitLab, and Slack. To mitigate this, we developed unit tests and manually checked results for accuracy with CQSE engineers. As we rely on an automatic rerun mechanism we may underestimate flaky test failures as a test may persistently fail even on reruns, thus overestimating true failures. However, given the less frequent occurrence of flaky failures, this overestimation should be relatively small, keeping our study's overall conclusions unchanged.

VII. Related Work

In the past decade, numerous studies have investigated various phases of the flake test lifecycle. In the following, we discuss related work that explores different flaky test cost factors and outline the research gap this paper aims to address.

a) Cost for Test Case Rerunning: Research at Google indicates that 16% of tests are flaky, consuming 2%-16% of computation resources for reruns [10, 11]. An et al. report that in the CI testing process of SAP HANA, 10% of testing time is allocated for reruns to address flakiness [40]. Parry et al. use machine learning and rerunning for flaky test detection. They estimate that 2,500 runs of the Apache Airflow test suite would cost 38,77 USD on an Amazon Web Services instance [39].

b) Cost for Investigating Test Failures: In their survey, Parry et al. summarize from two studies [56, 57] that there are many "false alarms" and thus developers might look for bugs that do not exist. This might lead to a "considerable waste of a developer's time" [7]. Aside from meta-studies, developer surveys are a repeatedly chosen method for obtaining information about the elusive costs of investigating test failures. Gruber and Fraser report wasted time as the most severe consequence of flaky tests perceived by developers. They do not specify at which part of the development process the time is wasted. With their developer survey, Habchi et al. come to the same conclusion [26]. In addition, they report that time is spent investigating the root cause of a test failure. Parry et al. find that developers perceive differentiating between deterministic and flaky failures differently: while some find it hard, others do not. Herzig et al. report that investigating a test failure, regardless of whether it was a flaky or deterministic failure, costs Microsoft an average of \$9.60 [42].

c) Cost for Repairing Flaky Tests: Costa et al. equate ticket comments about test flakiness with the resolution cost, finding greater comment counts imply higher complexity and cost [58]. Again, surveying developers is a commonly chosen tool to estimate the effort to repair flaky tests: In their survey, Eck et al. report that flaky tests are perceived as time-consuming since they are not easy to reproduce. They further indicate that flaky tests are often hard to fix and therefore "require a certain level of knowledge to be able to fix them" [13]. Parry et al. also report that most developers perceive reproduction of flaky failures as difficult, which makes it harder to repair these test [6].

d) Cost for Development Delays: Gallaba et al. found that 47% of the manually restarted builds on Travis CI passed without changes, suggesting failures due to flaky tests [44]. Parry et al. summarize from this study that "flaky tests can threaten the efficiency of CI by intermittently failing builds and requiring manual intervention, hindering the process of merging changes and stalling a project's development" [7].

e) Cost for Flaky Test Management: Lam et al. give an insight into Microsoft's CloudBuild system with its flaky test management system called Flakes [21]. This tool supports the four phases of detection, reporting, suppression, and resolution. However, no costs regarding Flakes are reported.

f) Cost for Production Bugs: Even though flaky tests are often implicitly assumed to have their root cause in the test code, Rahman and Rigby show that Firefox CI builds containing failing flaky tests receive more crash reports than average builds [48]. Shull et al. establish heuristics on the cost to fix a defect based on the opinion of a group of experts [50]: For severe defects, the cost for fixing is about 100 times higher than in the early phases of the development. For non-severe defects, the cost is still twice as high.
In summary, we are not aware of any prior work that compiles cost factors into a cost model for flaky tests in CI and quantifies these costs in an industrial-scale CI development process.

VIII. Conclusion

Flaky tests pose a significant challenge in software development. Despite the efforts of researchers and practitioners to mitigate the negative consequences of flaky tests, it is unclear what the actual cost drivers of flaky tests are.

In this study, we take a first step towards a holistic view of the costs of flaky tests in CI by compiling cost factors from research and practice into a cost model that depends on context-specific input parameters. The model aims to support practitioners in analyzing and understanding the origin of costs and in making decisions about where to invest effort. We conduct a case study in a large industrial software project at CQSE, where we quantify the total cost invested in flaky tests. Our findings reveal that flaky tests in this project account for at least 2.5% of the productive developer time in this particular context. This effort can be broken down into (1) investigating failed pipelines, (2) repairing flaky tests, and (3) managing flaky tests. Interestingly, we find that rerunning test cases, a common industry strategy to mitigate flaky test problems, incurs relatively negligible costs, accounting for 0.63% of all test execution costs in CI. Due to the low cost of computing time, this equates to \$3 per month for the project studied. The study results served as a basis for the decision to shift efforts from investigating or repairing to rerunning tests since the associated execution costs are comparatively inexpensive. The implementation of this strategy in the studied project successfully detects flaky failures that would have been missed with the previous strategy in 4.8% of all pipelines (i.e., every 20th pipeline would have failed due to a flaky test) and thus reduces the cost of investigating pipeline failures.

We encourage researchers and practitioners to use the cost model to analyze costs in other contexts and contribute to a better understanding of the costs of flaky tests.

REFERENCES

[1] H. Leung and L. White, "Insights into regression testing," in *Proceedings. Conference on Software Maintenance - 1989*. Miami, FL, USA: IEEE Comput. Soc. Press, 1989, pp. 60–69. [Online]. Available: http://ieeexplore.ieee.org/document/65194/

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://doi.wiley.com/10.1002/stv.430

[3] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong China: ACM, Nov. 2014, pp. 235–245. [Online]. Available: https://dl.acm.org/doi/10.1145/2635868.2635910

[4] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Beijing China: ACM, Jul. 2019, pp. 101–111. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3330570

[5] M. Harman and P. O'Hearn, "From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2018, pp. 1–23.

[6] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Surveying the developer experience of flaky tests," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 253–262.

[7] ——, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Jan. 2022.

[8] M. Gruber and G. Fraser, "A survey on how test flakiness affects developers and what support they need to address it," *IEEE Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2022.

[9] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. Seoul South Korea: ACM, Jun. 2020, pp. 110–119. [Online]. Available: https://dl.acm.org/doi/10.1145/3377813.3381370

[10] John Micco, "The State of Continuous Integration Testing @Google," 2017. [Online]. Available: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45880.pdf

[11] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 2019, pp. 101–110. [Online]. Available: https://ieeexplore.ieee.org/document/8804429/

[12] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Nov. 2014, pp. 643–653.

[13] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Aug. 2019, pp. 830–840.

[14] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An empirical study of flaky tests in Python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2021, pp. 148–158.

[15] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: automatically detecting flaky tests," in *Proceedings - International Conference on Software Engineering*, vol. 2018-January. IEEE Computer Society, 2018, pp. 433–444.

[16] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, "Practical flaky test prediction using common code evolution and test history data," Dublin, Ireland, pp. 210–221, Apr. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10132283/

[17] J. Morán, C. Augusto, A. Bertolino, C. D. L. Riva, and J. Tuya, "FlakyLoc: flakiness localization for reliable test suites in web applications," *Journal of Web Engineering*, Jun. 2020. [Online]. Available: https://journals.riverpublishers.com/index.php/JWE/article/view/3361

[18] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, "Flakycat: predicting flaky tests categories using few-shot learning," Melbourne, Australia, pp. 140–151, May 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10174109/

[19] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "IFixFlakies: a framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 545–555. [Online]. Available: https://doi.org/10.1145/3338906.3338925

[20] S. Dutta, A. Shi, and S. Misailovic, "FLEX: fixing flaky tests in machine learning projects by updating assertion bounds," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 603–614. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3468615

[21] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Jun. 2020, pp. 1471–1482.

[22] E. Fallahzadeh and P. C. Rigby, "The impact of flaky tests on historical test prioritization on Chrome," *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, vol. 10, pp. 273–282, 2022.

[23] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of UI-based flaky tests," in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, May 2021.

[24] N. Hashemi, A. Tahir, and S. Rasheed, "An empirical study of flaky tests in Javascript," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2022, pp. 24–34. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSME55016.2022.00011

[25] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, Nov. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3428270

[26] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Valencia, Spain: IEEE, Apr. 2022, pp. 244–255.

[27] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 337–348. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397359

[28] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in Android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 534–538.

[29] C. Ziftci and D. Cavalcanti, "De-flake your tests: automatically locating root causes of flaky tests in code at Google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2020, pp. 736–745.

[30] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. F. Bissyandé, "Peeler: Learning to effectively predict flakiness without running tests," in *2022 IEEE international conference on software maintenance and evolution (ICSME)*, 2022, pp. 257–268.

[31] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 1572–1584. [Online]. Available: https://ieeexplore.ieee.org/document/9402098/

[32] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," pp. 1912–1927, 2023.

[33] R. Wang, Y. Chen, and W. Lam, "iPFlakies: a framework for detecting and fixing python order-dependent flaky tests," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Pittsburgh, PA, USA: IEEE, May 2022, pp. 120–124. [Online]. Available: https://ieeexplore.ieee.org/document/9793801/

[34] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 301–311. [Online].

Available: https://ieeexplore.ieee.org/document/9240694/

[35] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Traon, "What made this test flake? Pinpointing classes responsible for test flakiness," Los Alamitos, CA, USA, pp. 352–363, Oct. 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSME55016.2022.00039

[36] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. Coimbra, Portugal: IEEE, Oct. 2020, pp. 403–413. [Online]. Available: https://ieeexplore.ieee.org/document/9251071/

[37] H. Leung and L. White, "A cost model to compare regression test strategies," in *Proceedings. Conference on Software Maintenance 1991*. Sorrento, Italy: IEEE Comput. Soc. Press, 1991, pp. 201–208. [Online]. Available: http://ieeexplore.ieee.org/document/160330/

[38] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. Virtual Denmark: ACM, Jul. 2021, pp. 553–565. [Online]. Available: https://dl.acm.org/doi/10.1145/3460319.3464840

[39] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Software Engineering*, vol. 28, no. 3, p. 72, May 2023. [Online]. Available: https://link.springer.com/10.1007/s10664-023-10307-w

[40] G. An, J. Yoon, T. Bach, J. Hong, and S. Yoo, "Just-in-Time Flaky Test Detection via Abstracted Failure Symptom Matching," Oct. 2023.

[41] C. Jordan, P. Foth, A. Pretschner, and M. Fruth, "Unreliable test infrastructures in automotive testing setups," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 307–308. [Online]. Available: https://dl.acm.org/doi/10.1145/3510457.3513069

[42] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 483–493. [Online]. Available: http://ieeexplore.ieee.org/document/7194599/

[43] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: when and what should we control?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 55–65. [Online]. Available: http://ieeexplore.ieee.org/document/7194561/

[44] K. Gallaba, M. Pinzger, C. Macho, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of Travis CI," in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, Sep. 2018, pp. 87–97.

[45] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Buenos Aires: IEEE, May 2017, pp. 233–242. [Online]. Available: http://ieeexplore.ieee.org/document/7965447/

[46] J. Palmer, "Test flakiness - methods for identifying and dealing with flaky tests," 2019. [Online]. Available: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

[47] J. Raine, "Reducing flaky builds by 18x," 2020. [Online]. Available: https://github.blog/2020-12-16-reducing-flaky-builds-by-18x

[48] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 857–862. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3275529

[49] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at Mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 1381–1392. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3473931

[50] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of the 8th International Symposium on Software Metrics*, ser. METRICS '02. USA: IEEE Computer Society, 2002, p. 249.

[51] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Denmark: ACM, Jul. 2021, pp. 491–504. [Online]. Available: https://dl.acm.org/doi/10.1145/3460319.3464834

[52] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "The importance of discerning flaky from fault-triggering test failures: a case study on the Chromium CI," Feb. 2023. [Online]. Available: http://arxiv.org/abs/2302.10594

[53] J. Micco, "Flaky tests at google and how we mitigate them," 2016. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[54] T. A. D. Henderson, B. Dorward, E. Nickell, C. Johnston, and A. Kondareddy, "Flake aware culprit finding," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023, pp. 362–373.

[55] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong China: ACM, Nov. 2014, pp. 19–29. [Online]. Available: https://dl.acm.org/doi/10.1145/2635868.2635892

[56] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, Sep. 2015, pp. 101–110. [Online]. Available: http://ieeexplore.ieee.org/document/7332456/

[57] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. San Jose, CA, USA: ACM Press, 2014, pp. 385–396. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2610384.2610404

[58] K. Costa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, pp. 1–14, 2022.