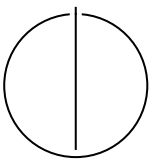
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Using Readily Available Information for Detecting Flaky Failures in Continuous Integration

Michael David Kuckuk



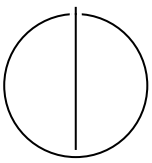
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Using Readily Available Information for Detecting Flaky Failures in Continuous Integration

Author:	Michael David Kuckuk
Examiner:	Prof. Dr. Alexander Pretschner
Supervisor:	Fabian Leinen
Submission Date:	March 6, 2026



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

Munich, March 6, 2026 Michael David Kuckuk

Acknowledgments

TODO: Acknowledgments

Abstract

TODO: Abstract

Contents

1 Introduction

In Continuous Integration (CI), automated tests are commonly used as a gating mechanism for accepting new code changes. Ideally, test outcomes are deterministic: given a specific test and a specific System under Test (SUT), all executions should yield the same result. In practice, however, some tests exhibit non-deterministic outcomes due to uncontrolled aspects of the environment (like random number generation or timing). Such tests are generally referred to as *flaky*, although definitions vary across the literature [1, 2].

In CI, failures block the process from continuing (e.g. a branch being merged or a new version being deployed), since a failure is assumed to indicate a fault in the system that should be fixed before the process can continue. Flaky failures break this assumption, which is why many projects seek to ignore them and allow the process to continue if all encountered failures were flaky. However, doing so manually still has many issues. The time involved in manual investigation can slow down development [3], investigation not only binds resources but also costs money [4], and frequent flakiness can deteriorate testing practice overall [5–7]. In addition, many developers report frustration with flakiness and perceive it as a waste of time [5, 7]. Unfortunately, flakiness is very common. Depending on the project, the percentage of builds impacted by flakiness can range from 0.4% to 35.7% [8].

This raises the need for automatic approaches to flaky failure detection. One common approach is to rerun failed tests multiple times and treat differing outcomes as evidence of flakiness, which leads to the initial failure being ignored [5, 7]. While rerunning is still cheaper than manual investigation [4], its cost can still build up. Furthermore, rerunning takes time and delays test results, which slows the development process.

Other approaches aim to identify flaky tests rather than flaky failures [9, 10]. If such a known flaky test fails, its failure can then be ignored. However, just because a test is flaky doesn't mean all of its failures must be caused by flakiness. A study on Chromium finds that one third of regression faults could be revealed by flaky tests [11]. Therefore, simply ignoring failures if the test is known to be flaky would miss many actual regressions.

Since flaky failures are known to have certain causes, there has also been research into using these causes to detect flaky failures [1, 12]. For example, DeFlaker [13] uses code coverage to detect flaky failures. However, this approach requires instrumenting

the code to collect coverage information, which adds overhead and complexity to the CI setup. For some project environments, this might require significant effort and make the approach impractical. Many other root-cause-based approaches suffer from similar issues.

Some research has investigated using machine learning for flakiness detection. For example, Gruber et al. [14] predict whether a test is flaky based on the test's flip rate, the number of changes to source files in the last 54 days, as well as the number of changed files in the most recent pull request. They achieve promising results, with an F1-score of 95.5%. However, their dataset is very limited. Furthermore, their approach suffers from the same applicability issues as other flaky test detection approaches, as explained above.

Hoang et al. [15] report similarly good results when using machine learning to classify flaky failures. However, they do not publish many details about their approach. In contrast to Gruber et al., they also include training data that can be harder to obtain in CI environments, such as RAM usage during the test's execution or the precise error message. Since machine learning is ultimately based on the statistical properties of the training data, it is unclear for how long these properties will remain similar, i.e. for how long the model will uphold its performance.

Our goal is to develop a practical approach to flaky failure detection. We use simple machine learning models TODO: currently only random forests due to the promising results they have shown in previous research. Furthermore, we limit ourselves to using information that is readily available in typical CI environments, such as historical outcomes, execution durations, and code evolution metadata. This allows easy adoption by developers, regardless of the specific programming language, framework, or CI system used in their project. As our ground truth, we use not only the results of reruns within the same job but also the results of reruns in fresh CI jobs. Using this ground truth, our model has the opportunity to detect more types of flakiness than immediate rerunning alone. Furthermore, we propose how to apply our approach in practice, accounting for expected changes in the test suite over time that would otherwise cause the performance of the model to degrade.

To evaluate this approach, we use a large dataset by Leinen et al. [16]. We investigate how well our approach can detect flaky failures as well as how well it can allow pipelines to pass that would have otherwise failed due to flaky failures. Since we propose a new process for flakiness detection, we also compare the cost of our approach to that of immediate rerunning.

2 Background

Before explaining our approach to flaky failure prediction, it is important to define and explain some key concepts that will become relevant in the following chapters. Software testing and CI are the foundation of this thesis, and flaky tests are the central concept that we will be working with. Machine learning classification is the core of our detection approach, and we will cover the basics of this topic as well.

2.1 Software Testing

Software testing is the process of executing a program with the intent of ensuring it behaves correctly and detecting failures CITATION NEEDED. Automated tests are pieces of code that exercise the SUT and compare its actual behavior to the expected behavior. When the actual behavior deviates from the expected behavior, the test fails.

Automated tests can be categorized by their scope. **Unit tests** test individual components (such as functions or classes) in isolation CITATION NEEDED. **Integration tests** verify that multiple components work correctly together. **End-to-end tests** (also called system tests) test the entire application from the user's perspective, often simulating user interactions with a graphical interface CITATION NEEDED.

There is a trade-off between these test types. Unit tests are typically fast to execute and easy to debug when they fail, but they cannot catch issues that only emerge when components interact. End-to-end tests can catch these issues but are slower, more complex to write, and more difficult to debug CITATION NEEDED.

If the main purpose of a test is to ensure that existing functionality isn't accidentally broken by new code, such tests are called **regression tests** [17].

2.2 Continuous Integration

CI is a software development practice where developers frequently integrate their code changes into a shared repository. Each integration is verified by an automated build, which typically includes running the automated test suite [18].

The main idea behind CI is to detect integration problems early. Instead of waiting until the end of a development cycle to integrate all components, developers constantly

have a fully integrated and working version of the code that they base their new changes on. They then integrate their changes back into the central version regularly. This makes it easier to identify which change introduced a bug and reduces the complexity of resolving merge conflicts [18].

In the context of CI, automated regression tests represent a gating mechanism for accepting new code changes. A test failure signifies that either the new code changes break existing functionality or that it introduces a completely new bug. In both cases, the issue needs to be resolved before the code changes can be accepted and integrated into the central version.

2.3 Flaky Tests

There are many different definitions for flaky tests [1]. They generally agree that flaky tests are automated tests that can both pass and fail when there haven't been any significant changes to the code.

This non-deterministic behavior can have various causes. For example, **order-dependent tests** are tests that pass when executed in some orders but fail when executed in other orders [19]. Another example is **test-case timeouts**, which can cause a test to pass if it finishes within the timeout, but fail if it takes too long [6], e.g. because the test runner is under more load than usual.

Some definitions are very restrictive, only considering tests as flaky if the cause for the non-deterministic behavior meets certain criteria [20]. Most definitions allow for any cause for non-determinism and only require the test code and the SUT to remain unchanged between the passes and fails [1].

We will use an even less restrictive definition, following Harman et al. [21]. This definition only requires the observation of different outcomes while all environmental factors that the tester seeks to control remain unchanged. This definition suits practical use cases better, as it matches more closely with what practicioners find annoying, namely test executions that should have been the same *by their own standards* but yet weren't.

2.3.1 Root Causes of Flaky Tests

TODO: Explain the different root causes of flaky tests

noch zu checken:

- [22]
- [23]

- [6]

2.3.2 The Cost of Flaky Tests

TODO: Flaky tests make PRs take longer to merge, the manual investigation binds resources, rerunning also costs time and resources, the flaky failures provide no real value

- study examined 4511 flaky job failures 25% of job failures are flaky, though only a fraction of those flaky job failures were caused by flaky test failures (they say 4% but their classification is different from ours and other categories might be considered flaky test failures by us or other research) [24]
- 9.8 - 16.3% of pipeline failures are caused by flaky failures that aren't detected by immediate rerunning [16]
- 99.63% of the cost caused by flaky failures in CI is diagnosis cost [24]
- developers worry about the loss in trust in test results [7]
- "Our analysis also suggests that developers who experience flaky tests more often may be more likely to ignore potentially genuine test failures." [25]
- flakiness leads to developers ignoring tests, which can lead to actual regressions being missed [5]
- A developer survey finds that 77% of developers consider flaky tests as less reliable and are more likely to ignore them [6]
- one study applies a cost model and finds that rerunning one test costs 0.02$ while letting it cause the pipeline to fail and having a developer investigate the failure costs 5.67$. in their study, this amounts to 1.1% of the total development time. Another 1.3% of development time is used for repairing flaky tests[4]
- in a survey, some developers report occasions when fixing flaky tests is not an option due to a lack of time and resources, which leads to them being ignored [5]
- developers mistake non-determinism in the SUT for non-determinism in the test. both cause flaky failures, but in the prior case, those failures are caused by actual bugs, not just by flaky tests [5]
- flaky failures deteriorate testing practice overall. first, developers can become afraid to introduce new flaky tests, which can lead to them writing fewer tests

overall. second, once there are sufficiently many flaky tests, developers might be more inclined to just accept more flaky tests, since the system already is flaky [5]

- manual investigation can cause the time until a PR is merged to slow down by a factor of three [3]

Noch zu checken:

- [13]
- [1]

2.3.3 Flaky Failures

Once practitioners have identified flaky tests, one action they might take is to ignore failures of these tests. This practice, however, is only sound if those flaky tests never fail due to actual regressions. However, Haben et al. [11] found that flaky tests also have high capabilities of revealing actual regressions. This means that simply detecting flaky tests and ignoring their failures under the assumption that the failure is due to flakiness is not a good idea. Instead, we should aim to identify flaky failures.

Therefore, similar to our definition of flaky tests, we define flaky failures as failures of tests where there could also be a successful execution of the same test without any changes to the environmental factors that the tester seeks to control.

Detecting flaky failures is also much more relevant to practical applications. First, similar to the point above, the existence of flaky tests on its own is not an issue. It is the failures of those tests that cause the costs that have been layed out in 2.3.2. Focusing on a given failure and determining whether it is flaky or not is a much more direct approach to reducing the costs.

Second, flaky tests are very difficult to identify. Depending on the root cause of their flakiness, a flaky failure might only occur very rarely. An empirical study ran test suites 10,000 times and yet didn't manage to encounter any failures for some previously identified flaky tests [10]. When analyzing failures, however, one must only produce a successful run, which is much more likely to happen. Identifying given failures as flaky doesn't have this problem. One study found that a failure can be classified as flaky or not flaky with a confidence of 95% by rerunning the failed test up to 3 times [26].

2.3.4 Mitigation Strategies

- siehe background von [7]
- Rerunning most common mitigation strategy [7]

- Automatic rerunning is natively supported in some project environments, which leads to more manual and thus more costly mitigation strategies [7]

Noch zu checken:

- [6]

2.4 Machine Learning Classification

2.4.1 Binary Classification

2.4.2 Features

2.4.3 Common Classifiers

2.4.4 Evaluation Metrics

2.4.5 Class Imbalance

3 Related Work

There has been a lot of research into flaky test detection in the past. Many approaches focus on specific causes for flakiness, such as concurrency or timeouts. Others focus language instrumentation, vocabulary, or code evolution. There have also been some approaches based on machine learning.

These approaches have various limitations. Some are very involved to set up, such as language instrumentation or code coverage. Most compare themselves with a baseline of immediate reruns, which is not sufficient to detect all flaky tests.

TODO: [13] analyze flaky failures to identify flaky tests

- [27] sagen dass rerunning state of the art ist um failures zu klassifizieren
- flaky failures are often repetitive. error messages and stack traces can achieve a 100% specificity in some projects [27]. Note: but they're a tad bit more complicated to obtain
- vocabulary-based machine learning with good results [9]
- machine learning with static features with good results compared to envolved approaches [28]
- FlakeFlagger classifies flaky tests using machine learning + bag of words [10]
- [29] detect flaky builds using machine learning and log-based features

noch zu checken:

- [13]
- [10]
- [23]

3.1 Common Causes of Flakiness

3.2 Causal Detection

3.3 Code Evolution-Based Detection

3.4 Machine Learning-Based Detection

4 Approach

In order to identify flaky failures in CI, we devleop a machine learning approach that can classify a given test failure as either flaky or non-flaky based on readily available information, such as historical test execution data and code change metadata. In order to identify as many flaky failures as possible, we compute our ground truth using immediate reruns as well as validation runs. Since code bases change over time, we retrain our model regularly to adapt to these changes. With our model, we can then determine whether all failures of a given pipeline are flaky, in which case the pipeline can be salvaged (i.e. it passes despite the failed tests).

4.1 Prerequisites

Using this approach only makes sense in projects that meet certain requirements. First, they must have a CI environment that automatically runs the project's test suite. Naturally, the test suite should frequently run into flaky failures in order for the effort of implementing this approach to be worthwhile.

Second, every CI run must be associated with a specific version of the code in a Version Control System (VCS). The VCS must be able to provide information about how the code has developed over time, such as the names of files changed since the last run.

These systems must be readily available and provide certain information for our features to be computed from. This must be the case both during model training (which happens not just once but periodically) and during model usage (which happens for every failure).

While some of this information is only related to the current failure being evaluated, some information on the history of the test at hand is also required. We therefore also require access to information about failures for past code versions.

For each failure, the CI system must must provide the following information:

- the result of the test execution (pass or fail)
- the execution duration of the test execution
- the name of the test

- what version of the code was used for the test execution

This information must not only be available for the current failure, but also for past failures as it will be needed to compute features based on e.g. the past 1,000 code versions. If this information is not natively available, it must be manually collected and stored (more on this in Subsection 4.4.2).

The VCS must then be able to provide the following information:

- the ancestor versions of any version of the code that was used for a test execution
- the paths to all files that were changed between any two versions of the code

Note that Git[1], for example, doesn't necessarily fulfill these requirements. If a project's branching strategy includes history rewriting, such as squash merges, then Git's history doesn't properly reflect the actual history of the code. In such environments, other systems would be needed to accurately track the code history.

Some of the features we compute require information about the previous 1,000 code versions. While these features can be computed on demand for every failure, it may be more efficient to cache some history-related information for code versions that are expected to be the ancestors of future code versions. Therefore, having a persistent storage for this information can be beneficial.

Finally, it must be possible to (conditionally) rerun failed tests both in the same CI job as well as in some fresh job. This is necessary to compute the ground truth for training.

4.2 Labeling Strategy

To train our model, we first need to establish a ground truth for the failures encountered in the CI environment. In order to increase the amount of flaky failures that we can detect, we require the ground truth to contain more than just regular reruns.

Following the definitions by Leinen et al. [16], we distinguish between two types of reruns: an **immediate rerun**, which is a re-execution of the test within the same CI job, and a **validation run**, which is a re-execution in a fresh CI job. While immediate reruns are faster and thus cheaper to execute, they struggle to detect some flaky failures. Leinen et al. found that validation runs are capable of detecting significantly more failures and are thus able to recover more pipelines.

Therefore, we require our ground truth to consist of both immediate reruns and validation runs. To determine the label for a failure, we execute one immediate rerun

[1] https://git-scm.com/

and, if it also fails, one validation run. If either of these reruns is successful, we classify the failure as flaky. Otherwise, if both reruns fail, the failure is classified as non-flaky.

To collect training data for a specific time period (discussed further in Section 4.3), we gather the features described in Section 4.4 for every failure encountered by the CI system. To determine the labels for these failures, we execute immediate reruns and validation runs, as described above.

4.3 Operational Usage and Retraining

Once the model is trained, it is integrated into the CI pipeline's operational flow. When a pipeline encounters test failures, the model classifies each failure as either flaky or non-flaky. If all failures in a pipeline are classified as flaky, the originally failed pipeline is recovered, allowing development to proceed without prolonging the interruption by running immediate reruns, validation runs, or requiring any manual investigation. Conversely, if any failure is classified as non-flaky, the pipeline fails, signaling that manual investigation is required. This process is illustrated in Figure 4.1.

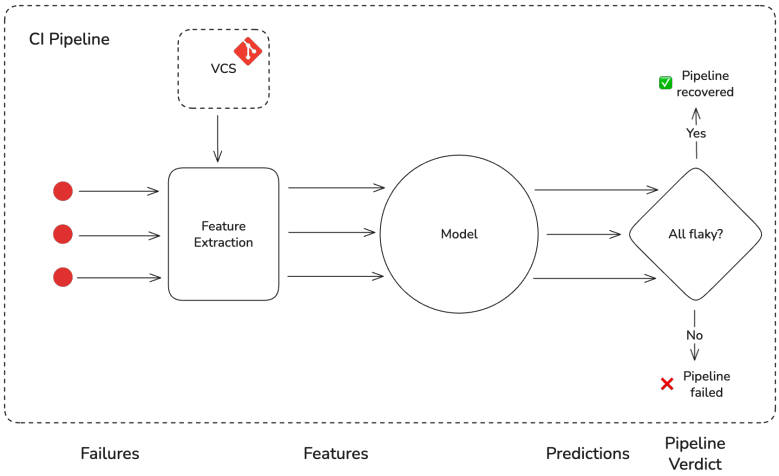


Figure 4.1: How the model is used to recover a CI pipeline that contains failed test runs.

However, the nature of flaky failures changes over time. Moriconi et al. found that the underlying causes of flakiness can shift significantly, with some of the most common causes they found overall not appearing for extended periods of time [30]. To address this information drift and ensure the model remains accurate, it must be retrained periodically.

To this end, practitioners must define two key intervals: the *training window size* and the *usage window size*. The training window size specifies the duration for which ground truth data (i.e. a immediate rerun and a validation run) is accumulated. The usage window size defines how frequently the model is retrained. Figure 4.2 illustrates the relationship between these windows.

If the usage window is larger than the training window (Figure 4.2b), the system can operate for periods without requiring new training data, i.e. neither immediate reruns nor validation runs need to be executed for any failures that occur during this time.

In contrast, if the usage window is smaller than the training window (Figure 4.2a), data collection must be continuous, requiring immediate reruns and validation runs for all failures. Despite this, the model remains the primary decision-maker for the pipeline's outcome. This allows the resource-intensive reruns to be decoupled from the developer's workflow, as they can be executed asynchronously while the model provides an immediate verdict. In addition to allowing development to proceed with less of an interruption, this also allows validation runs to be executed during off-peak hours, e.g. when CI load or electricity costs are low or when renewable energy is more readily available.

4.4 Feature Engineering

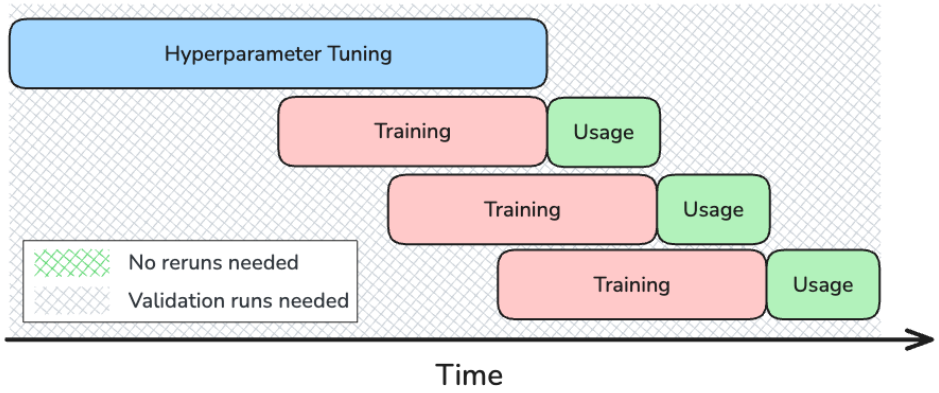
The features used by our model are all derived from information we can easily obtain from the CI system. Despite its high availability, there are some considerations to be made for how exactly to obtain and represent various information.

4.4.1 Dealing with Code Evolution History

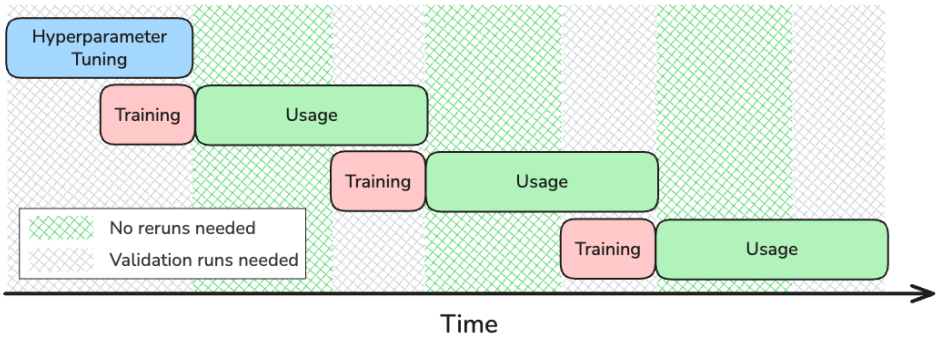
CI pipelines are triggered based on specific versions of the code. Analyzing how tests and their failures have behaved in and evolved throughout different versions can yield valuable insights for predicting flakiness.

Following Alshammari et al. [10], we therefore represent history in units of commits. This stands in contrast to representing history as regular timestamps [14]. We chose this approach because it better reflects development settings in which many changes happen concurrently and independently from each other. Timestamps alone cannot distinguish between such concurrent changes, and also cannot properly reflect when new changes are based on a very old version of the code.

However, this acceptance of non-linear history comes with a challenge. Many features are trivial to compute for linear histories (e.g. the percentage of failed test runs). However, for a merge commit of two branches, this is not clearly defined. If both parents have various commits with test executions, should each execution have the



(a) Training window larger than usage window



(b) Training window smaller than usage window

Figure 4.2: Sketch showing the retraining process. After an initial phase of hyperparameter tuning, the model is trained and then used for the pre-defined interval sizes. Toward the end of the usage period, the model is retrained again in order for a new model to be available for the next usage period. The sketch visualizes the effect of different training and usage window sizes on the amount of immediate reruns and validation runs that need to be executed.

same influence on a feature? Should one of the parents have more influence than the other? What if the test only was introduced in one of the parents and no information for it exists in the other parent?

To address this issue, for any given code history, we compute the set of all possible **linearizations** of that history by starting at the root commit and descending into its parents. For each parent, we recursively compute its linearizations and then combine each of its linearizations with the current commit to form the linearizations of the

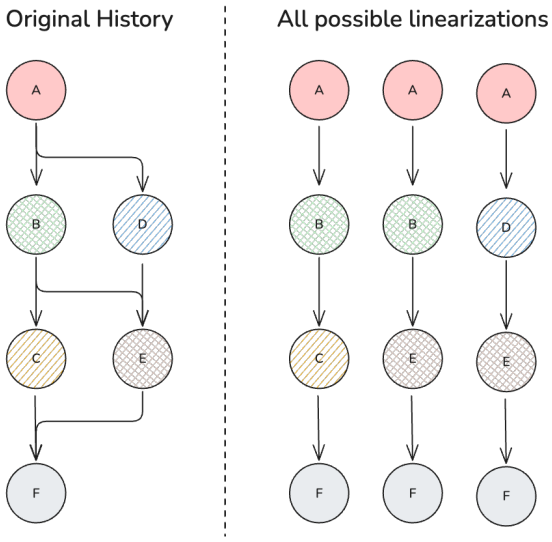


Figure 4.3: An example of how a code history with merges would be turned into a set of linear histories for feature extraction.

current commit. This process is illustrated in Figure 4.3.

In order to compute features defined on linear histories, we compute the features for each linearization and then aggregate the results into single values by taking the mean, minimum, and maximum of the feature values. These three values then become three different versions of the feature. They can then be reduced down during feature selection, if they are sufficiently redundant, or made available to the model as separate features in order for it to learn more complex relationships.

4.4.2 Computing Features Based on History

Following Hoang et al., who report improvements when considering multiple time frames for their data collection [15], we collect our history-based features for multiple **failure history windows**. To supply our model with a sufficient amount of detail, we choose failure history window sizes of 2, 5, 10, 50, 100, 200, 500, and 1,000 commits. If fewer commits are available than required by a window size (e.g. for newly introduced tests), the corresponding feature value for that failure history window is set to missing.

In order to reflect that recent developments may be more significant than older ones for some features, we apply various decay functions to the history-based features, creating one variant of the feature for each decay function. Following Gruber et al. [14], we use constant, reciprocal, and reciprocal-squared decay, as they yielded the most promising results in their experiments.

It is important to note that our failure history window sizes can exceed the amount of time that the model is trained or used for. This allows the model to obtain any context that it deems relevant for the current failure. And due to our limitation to readily available information, this choice doesn't require any additional effort to be made, i.e. no extra information needs to be collected (unlike during training periods, as described in Section 4.2). It's only important that this historical information is available to be collected, as mentioned in Section 4.1.

As an alternative to collecting this information and computing features based on it retroactively, we could also collect it live, i.e. after every pipeline run, and then store cached intermediate results for each commit. While this requires additional implementation effort, it increases the speed at which features can be computed when a new failure occurs that requires information from old commits.

TODO: ensure that this clarifies we compute the features for succesful runs as well as failed runs

4.4.3 History-Based Features

To keep our feature definitions concise and independent of specific decay functions, we define the set of all possible decay functions as $\mathcal{W} := \{w : \mathbb{N}_1 \to (0, \infty]\}$. The features are then defined as functions that accept one such decay function as an additional argument.

Furthermore, we define the set of all possible test execution results as $\mathcal{R} := \{\text{pass}, \text{fail}\}$.

The features computed following this linearization-based approach are thus the following:

Failure Rate: The rate at which this failure's test has failed in the past, taken from Hoang et al. [15] and adopted to our concept of history and extended with decay functions. The failure rate $fail\_rate : \mathcal{R}^n \times \mathcal{W} \to [0, 1]$ for a history of size $n \in \mathbb{N}_1$ is then computed as

$$fail\_rate(results, w) := \sum_{t=1}^{n} \frac{w(t)}{\sum_{u=1}^{n} w(u)} \cdot \begin{cases} 1 & \text{if } results_t = \text{fail} \\ 0 & \text{otherwise} \end{cases}$$

Flip Rate: The rate at which this failure's test results have flipped in the past, taken from Gruber et al. [14]. Note that flips describe pairs of consecutive commits, meaning a history of size n can have a maximum of $n - 1$ flips. TODO: add intuition for why this may be useful / what the hypothesis behind this feature is With the same definition of *results* as above, the flip rate $flip\_rate : \mathcal{R}^n \times \mathcal{W} \to [0, 1]$ is then computed as

$$flip\_rate(results, w) := \sum_{t=1}^{n-1} \frac{w(t)}{\sum_{u=1}^{n-1} w(u)} \cdot \begin{cases} 1 & \text{if } results_t \neq results_{t+1} \\ 0 & \text{otherwise} \end{cases}$$

Historical Duration: The duration of the failure's test in the past, taken from Hoang et al. [15] and adopted to our concept of history and extended with decay functions. Since flaky failures often take longer than non-flaky failures [15, 31], we compute the historical duration of all runs as well as the duration of only successful runs and only failed runs. Therefore, the three features $overall\_duration, failure\_duration, success\_duration : \mathbb{R}_{\geq 0}^{n} \times \mathcal{W} \to \mathbb{R}_{\geq 0}$ are defined as

$$overall\_duration(durations, w) := \sum_{t=1}^{n} \frac{w(t)}{\sum_{u=1}^{n} w(u)} \cdot durations_t$$

$$failure\_duration(durations, w) := \sum_{t=1}^{n} \frac{w(t)}{\sum_{u=1}^{n} w(u)} \cdot \begin{cases} durations_t & \text{if } results_t = \text{fail} \\ 0 & \text{otherwise} \end{cases}$$

$$success\_duration(durations, w) := \sum_{t=1}^{n} \frac{w(t)}{\sum_{u=1}^{n} w(u)} \cdot \begin{cases} durations_t & \text{if } results_t = \text{pass} \\ 0 & \text{otherwise} \end{cases}$$

Duration Difference: The difference between the duration of the failure at hand and the historical duration of the failure's test in the past, taken from Gruber et al. [14]. This yields the three features $overall\_duration\_diff$, $failure\_duration\_diff$, and $success\_duration\_diff : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}^{n} \times \mathcal{W} \to \mathbb{R}_{\geq 0}$ as

$$overall\_duration\_diff(orig, durations, w) := orig - overall\_duration(durations, w)$$
$$failure\_duration\_diff(orig, durations, w) := orig - failure\_duration(durations, w)$$
$$success\_duration\_diff(orig, durations, w) := orig - success\_duration(durations, w)$$

4.4.4 Failure-Based Features

In addition to history-based features, we compute features from the current failure context.

Pipeline and Job Failure Rate: The proportion of failed test runs in the same pipeline and in the same job as the current failure, following Hoang et al., who found that real failures often occur together while flaky failures tend to be more isolated [15]. The features $pipeline\_failure\_rate, job\_failure\_rate : \mathcal{R}^k \to [0, 1]$ for pipelines or jobs with k different test runs are defined as

$$pipeline\_failure\_rate(results) := \frac{1}{k}\sum_{t=1}^{k}\begin{cases}1 & \text{if } results_t = \text{fail}\\ 0 & \text{otherwise}\end{cases}$$

$$job\_failure\_rate(results) := \frac{1}{k}\sum_{t=1}^{k}\begin{cases}1 & \text{if } results_t = \text{fail}\\ 0 & \text{otherwise}\end{cases}$$

Distance to Changed Files: Following Hoang et al. [15], we would like to capture the distance between the failed test file and the files that were changed since the last known test execution. Since there can be multiple changed files, we compute the min, mean, and max distance between the test name and all of these changed file paths and provide all three values to the model as features. As our distance metric, we use the Levenshtein distance. TODO: intuition

Original Duration: The duration of the original test execution, following Hoang et al. [15] and Gruber et al. [14]. TODO: intuition

Weekday: The day of the week when the test was executed, ranging from 0 (Monday) to 6 (Sunday). Since development on weekends is usually less frequent, tests run on weekends may have different probabilities for failures to be flaky than those executed on weekdays. This feature allows the model to identify such patterns.

Test Name Vocabulary: We build a bag-of-words representation over test names in the training set, then create one feature per word, representing its frequency in the current test name. Vocabulary-based approaches have shown promising results in previous research due to certain terms being more common in flaky tests than in non-flaky tests [9, 10, 32].

4.4.5 Feature Reduction

Given the large feature space and the fact that many features are only slightly different variations of each other, we need to reduce the amount of features so that the model has a chance to find the most informative ones instead of being overwhelmed by the noise.

Starting with the vocabulary-based features, we drop features with information gain below 0.01 on the training set, following Alshammari et al. [10]. This likely reduces the potentially very large size of vocabulary-based features, keeping only those that are expected to be informative.

Next, we reduce the redundancy of the remaining features. To do so, we compute the information gain of each feature as well as the correlation between each pair of features. We then iterate through all features, starting with the one with the highest information gain, and remove all remaining features that have a correlation greater than 0.9 with

the current feature. TODO: consider adding pseudocode or a short algorithm box for the feature-reduction procedure.

4.5 Model Training

As random forests have yielded good results in previous research [10, 14] and are easy to use and interpret, we choose to use them as our classifier. Before training, we apply random undersampling to balance the classes.

Random forests require several hyperparameters, including the number of trees, the tree depth, and the number of features considered at each split. We assume that, for a fixed training window size, one hyperparameter configuration generally remains suitable over time. Based on this assumption, we tune hyperparameters only on the first window of each window size. We then reuse the configuration with the best F1 score for all later retraining steps with the same window size (see Figure 4.2). As opposed to regular training, this process also requires some validation data to be available. Given a training window of size w, we validate our hyperparameters on the immediately following data for a period of length $0.2w$.

We perform hyperparameter tuning with Optuna [33] on the following search space for the hyperparameters:

- Number of trees: between 50 and 500
- Maximum tree depth: between 5 and 75
- Minimum samples split: between 2 and 20
- Minimum samples leaf: between 1 and 20
- Maximum features per tree: one of sqrt, log2, 0.33, 0.5, or 0.8

5 Evaluation

In operational CI use, the central question is whether a failed pipeline can be safely recovered. Our approach is only useful in practice if it recovers pipelines that would otherwise fail due to flaky failures, while still not recovering pipelines that contain deterministic failures.

At the same time, the model itself is trained to classify individual failures, and most related work is also evaluated at test or failure level. To make our results as comparable to other work as possible and to understand the model's behavior directly, we therefore evaluate both levels: failure-level classification quality and pipeline-level recovery quality.

5.1 Research Questions

From the interests described above, we formulate the following research questions:

RQ1: How well does our approach detect flaky failures compared to immediate rerunning?

RQ2: How well does our approach recover pipelines from flaky failures compared to immediate rerunning?

RQ1 evaluates the classification task on the level of individual failures. This perspective is included because the model is trained on per-failure labels and we can thus gain the most direct insights into its behavior on this level. Furthermore, most related work is also evaluated at test or failure level, so this perspective is necessary to make our results as comparable to other work as possible (despite comparability being severely hindered by the different definition of ground truth used by our approach compared to most other work).

RQ2 evaluates the operational effect on the level that matters to developers: whether a blocked pipeline can be recovered. In our usage process (Section 4.3), a pipeline is recovered only if all its failures are classified as flaky. This makes pipeline recovery a stricter and practically more relevant criterion than per-failure classification alone. RQ2 therefore examines whether the approach can recover more pipelines than immediate

rerunning without falsely recovering non-flaky pipelines, allowing real regressions to go unnoticed.

5.2 Study subject

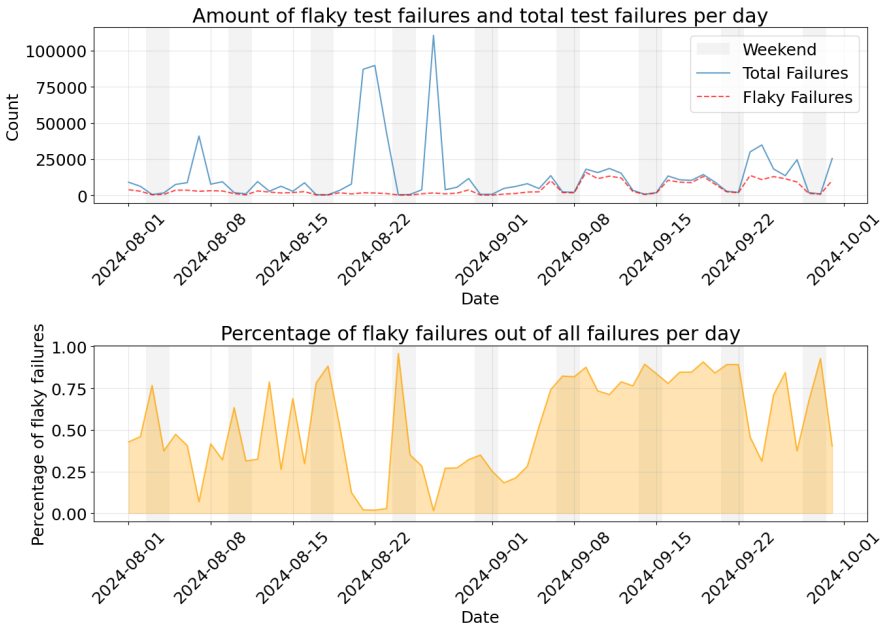


Figure 5.1: Distribution of flaky and non-flaky failures in GitLab's CI during August and September 2024, both in absolute numbers and as the percentage of flaky failures out of all failures.

We evaluate the approach on two months of CI data from GitLab (August and September 2024), using the dataset introduced by Leinen et al. [16]. This dataset is a good fit for our setting because (beyond containing the data we require for computing our features and the ground truth) it contains both pre- and post-merge pipelines and provides publicly available VCS information for the involved revisions.

In their CI, GitLab executes one immediate rerun for every failure they encounter. However, if too many failures occur at the same time, they employ a fail-fast strategy and skip immediate reruns.

For their dataset, Leinen et al. group an original test execution, its immediate reruns, and its validation run (if they exist) into a Test Execution Batch (TEB). Each TEB is labeled with one of several verdicts:

- *passed* - Passed on the first execution
- *failed* - Failed in all runs
- *flaky (detected)* - The test was detected as flaky by immediate rerunning
- *flaky (undetected)* - All immediate reruns also failed but a validation run detected the test as flaky
- *flaky (unchallenged)* - Validation runs were skipped (e.g. due to a fail-fast strategy) and a validation run detected the test as flaky
- *failed (unvalidated)* - Validation runs couldn't be executed for some reason.

Before running any computations, we discard all TEBs with the verdict *failed (unvalidated)*, because no reliable validation outcome is available for them.

Since our model only classifies failed executions, *passed* entries are mostly ignored. We only use entries with this verdict to compute some of our features (e.g. the failure rate). Since the verdict itself contains information about immediate reruns and validation runs, those features use other properties of *passed* entries (e.g. the result of the initial test execution).

For our ground truth, we label all TEBs with the verdict *failed* as non-flaky. TEBs with all other verdicts (i.e. *passed*, *flaky (detected)*, *flaky (undetected)*, and *flaky (unchallenged)*) are labeled as flaky.

Figure 5.1 shows the distribution of flaky and non-flaky failures over time. There are some spikes in deterministic failures (without corresponding flaky failures) as well as a clear change in the flakiness distribution around 2024-09-06. And even apart from these anomalies, the flakiness distribution is not stable, regularly shifting between about $\frac{1}{4}$ and $\frac{3}{4}$ flaky failures. These observations support the need for periodically retraining the model, as discussed in Section 4.3.

For feature computation, we first load the commit associated with each failure and up to 100 of its ancestors (via `git rev-list -max-count 101 <hash>`, where 101 includes the failure's commit itself). This choice follows the assumption that no more than 100 commits occur between two relevant CI executions. Using this VCS information, we identify the nearest ancestors with associated test runs and, following the history-construction process from Section 4.4, derive all feasible linear histories and compute our features accordingly.

Finally, the dataset includes job IDs and pipeline IDs for all TEBs, which allows us to evaluate not only failure-level classification, but also how many jobs and entire pipelines can be recovered under different configurations. Over the observed period, the share of

flaky and non-flaky failures is not stable, but shows noticeable shifts and short spikes (Figure 5.1). This variability motivates periodic retraining in the evaluations below.

TODO: include table with distribution from [16]?

5.3 Experiment Setup

Since our approach requires retraining the model periodically, our evaluation must necessarily take that into account. While we could also train a model on the entire dataset and then evaluate it on the entire dataset, those results would not be representative of the real-world usage of the model.

We therefore consider different training window sizes for our various evaluations. For each training window size, we train a model on all possible training windows of that size such that at least 20% of the training window size remains for evaluation. We then evaluate the model on every remaining day of the dataset. This is illustrated in Figure 5.2. The research questions are answered based on these evaluations, possibly disregarding some of the data e.g. if only specific usage window sizes are being evaluated and the models have been evaluated for longer than that.

To provide a wide range of training window sizes while still having enough test data, we consider training window sizes of 3 days, 1 week, 2 weeks, 3, weeks, 4 weeks, and 6 weeks.

Note that the longer a model is trained for and the later the training begins, the less data is available for evaluation (also observable in Figure 5.2). This causes a few phenomena in our evaluations. First, the longer the training window, the less days we are able to evaluate it after training. This makes the curves in some diagrams (e.g. Figure 5.3) shorter for longer training windows. Depending on the exact diagram, the curves can start later or end earlier than others (due to not being evaluated in earlier parts of the data, e.g. in Figure 5.7 or not being evaluated for days so long after training, e.g. in Figure 5.3, respectively). Second, this means that diagrams where every data point is an average over all models evaluated on that day, the amount of those models will vary depending on the training window size and the day. This can even be observed in curves that include standard deviations, where the standard deviation will become 0 for days that only one model could be evaluated on.

5.3.1 Flaky Failure Detection (RQ1)

To answer this question, we first analyze the models' performance after training to see how well they perform in general and how fast their performance degrades over time. We do this by grouping the models' predictions by day and computing the daily F1 score, precision, and recall. Note that we do this for each model trained on each

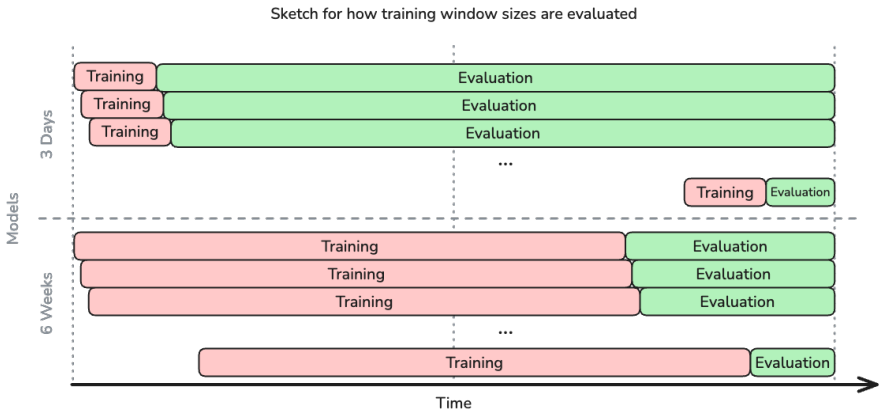


Figure 5.2: Sketch of the evaluation timeline. For each training window size, we train multiple models on all possible training windows of that size such that at least 20% of the training window size remains for evaluation. Note that proportions are not accurate in this sketch.

window, i.e. most days will be evaluated multiple times. We then group these results by training window size and day after training to achieve e.g. all metrics that models trained on some 3 day window achieved on the 6th day after their respective training. We then compute the mean and standard deviation of these metrics for each training window size and day after training.

Since the baseline (immediate rerunning) has no training window, we cannot compute its metrics per day after training. Instead, we compute the mean of the metrics over the entire period and use that for comparison to our models.

We also evaluate the models' performance across different usage window sizes (1 day, 3 days, 5 days, 7 days, 14 days). For each combination of training window size and usage window size, we train a model on every available window matching that configuration and compute the mean precision, recall, and F1 score that it achieves. We then compute the mean of those scores across all equal configurations to obtain a single value for each configuration and metric in order to aid comparison to other approaches.

TODO: document and explain heat maps

5.3.2 Pipeline Recovery (RQ2)

From a practical perspective, we are most interested in how many pipelines could be saved by our model in different configurations as opposed to immediate rerunning. Thus, we perform similar analyses as for the flaky failure detection, but transform the model predictions (and the immediate rerunning results) into a pipeline recovery

classification. Pipelines are recovered iff all failures in the pipeline are classified as flaky by the model (or the immediate rerunning).

To answer RQ2, we need to determine how many recoverable pipelines are correctly recovered (recall) as well as how many pipelines that we recover are actually recoverable (precision). While the prior is the motivating factor of this approach, since it is what unblocks development, the latter is important for quality assurance. Since false pipeline recovery means that actual faults go unnoticed, this metric is crucial for evaluating whether this approach can be used in a given project.

TODO: document and explain heat maps

5.4 Results

TODO: some transition

5.4.1 Flaky Failure Detection (RQ1)

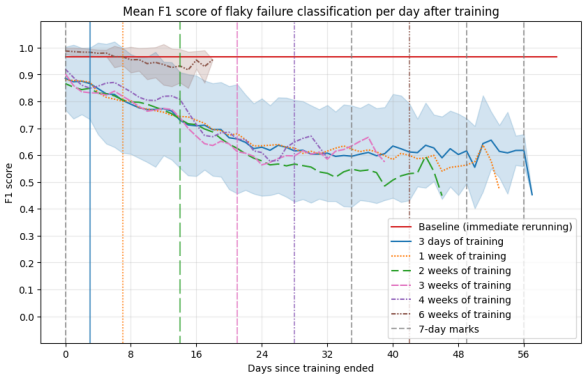
The results of analyzing the F1 score, precision, and recall of the flaky failure detection after training are shown in Figure 5.3. First, all three graphs show that the models perform no better on average than immediate rerunning, even on their first day of evaluation. One notable exception to this is the recall of the 6-week models, which will be discussed in more detail later.

Focusing on precision, we must first acknowledge that immediate rerunning achieves 100% precision by definition, since it only classifies failures as non-flaky if an immediate rerun is successful, which makes the failure flaky by definition. This level of precision is very difficult to achieve by a machine learning model and, in fact, our approach doesn't manage to achieve it with any training window size (judged by the average precision per day after training, determined as described in Subsection 5.3.1).

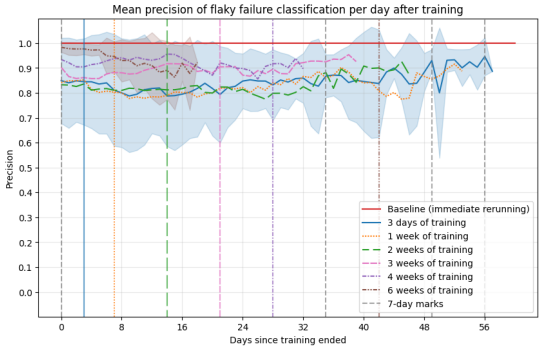
However, the precision scores achieved by our models do not seem to degrade over time. Recall, however, does degrade significantly (again, ignoring the 6-week models and leaving them for later discussion). The models trained on up to 2 weeks start off beating the recall of immediate rerunning, but this advantage disappears quickly. All models degrade at similar rates throughout the first four weeks before stabilizing at around 50% recall.

TODO: F1 irgendwie erwähnen, evtl mit anderen modellen vergleichen? oder als qualitätsmerkmal hernehmen?

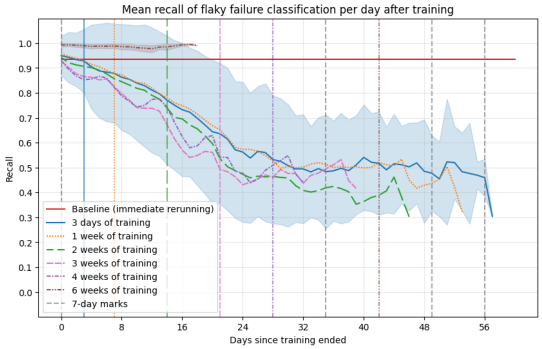
All three metrics seem to suggest that 6 days of training perform significantly better than all other training window sizes. However, as mentioned in Section 5.3, the portion of the dataset that could be used for evaluation differs for each training window size. Coincidentally, the portion of the dataset that the 6-week models are evaluated on is



(a) F1



(b) Precision



(c) Recall

Figure 5.3: Degradation of F1 score, precision, and recall of flaky failure detection over time for different training window sizes. For the edge cases (3 days and 6 weeks), the standard deviation has been added (the rest were omitted for better readability). Vertical lines matching the metric curves indicate when the model's usage time has exceeded its training window size.

significantly more stable than the portions of the other models (as can be seen when comparing the evaluation periods sketched in Figure 5.2 to the distribution of flakiness shown in Figure 5.1).

In order to be able to better compare the training window sizes, we perform the same analysis a second time. This time, we only consider models with evaluation windows matching one of the 6-week models (as shown in Figure 5.4). The results of this analysis are shown in Figure 5.5.

This perspective shows that, in this more equitable comparison of training window sizes, we find that the training window sizes perform much more similarly to each other. Longer training windows seem to perform slightly better than shorter windows, but

the difference is not as significant as in the analysis including all available evaluation data. Furthermore, we see vastly different values for this reduced evaluation data. This suggests that the results are very dependent on the exact evaluation data.

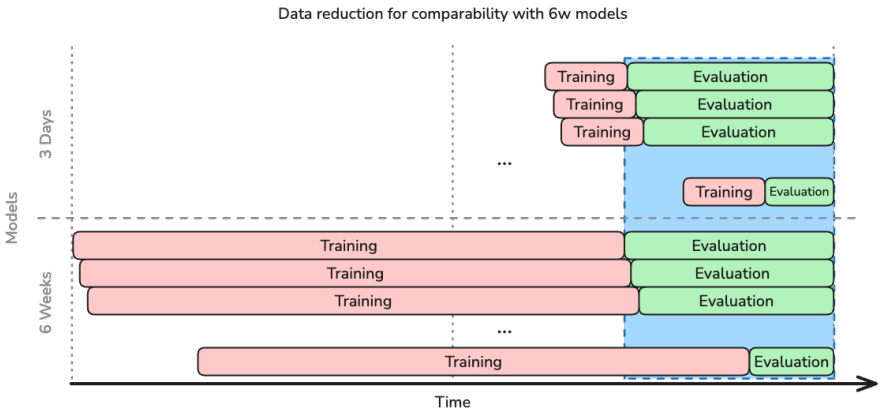
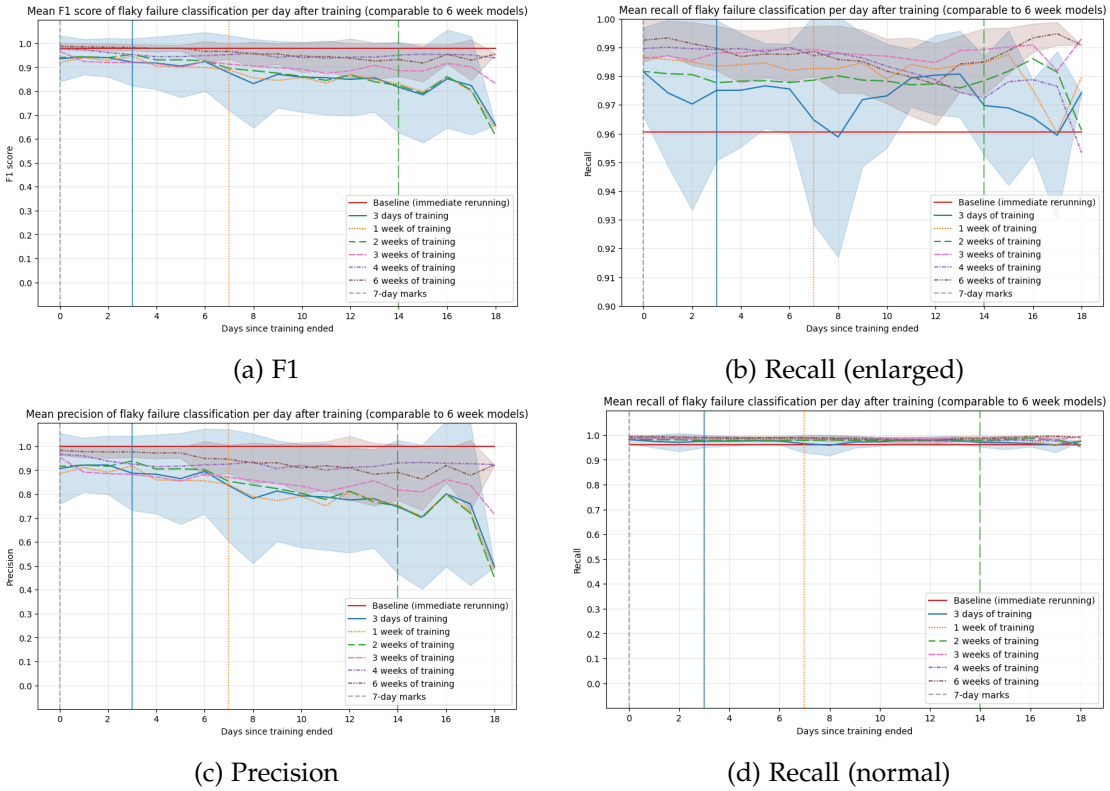


Figure 5.4: Sketch of the evaluation timeline when only considering models with evaluation windows matching one of the 6-week models.

Finally, we report average scores achieved for the aforementioned training and usage window sizes in Figure 5.6. Ignoring the 6-week models for reasons explained above, the best F1 score of 92% is achieved with 4 weeks of training and one day of usage. This configuration also achieves the highest precision of 93.4% as well as the highest AUPRC score of 98.1%. The highest recall of 95.8%, however, is achieved with only 1 week of training and 1 day of usage. When not ignoring the 6-week models, they perform best in all metrics when used for 1 day, with an F1 score of 08.8%, a precision of 98.3%, a recall of 99.3%, and an AUPRC of 99.9%.

When only considering models with evaluation windows comparable to the 6-week models, the scores improve significantly for all models. However, even in this more equitable comparison of training window sizes, the 6-week models still perform best with the configuration and metrics mentioned above. One notable observation is that, for this evaluation period, the 3-day models achieve a significantly worse AUC score than the other training window sizes, while the F1 scores remain comparable, suggesting that taking the model's confidence into account during classification could improve the model's performance.



(a) F1

(b) Recall (enlarged)

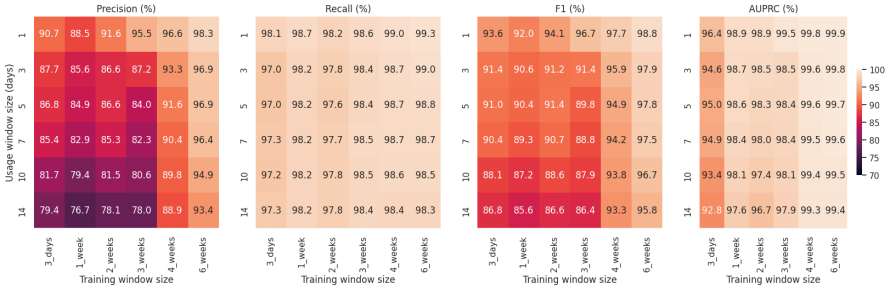
(c) Precision

(d) Recall (normal)

Figure 5.5: Degradation of F1 score, precision, and recall (as in Figure 5.3, but only considering models with evaluation windows matching one of the 6-week models). Since the recall curves are very close to each other, we have included another version of the same plot with an enlarged y-axis for better distinguishability of the curves.

Answer to RQ1: Based on our dataset, even within short usage periods, our approach is not able to beat immediate rerunning in classifying flaky failures as measured by F1 score, precision, or recall. It usually identifies less flakily failed tests and is unable to compete with the 100% precision that immediate rerunning achieves by definition. Recall degrades significantly during the first four weeks before stabilizing at around 50%. Precision does not degrade.

(a) All models



(b) Only considering models with evaluation windows matching one of the 6-week models

Figure 5.6: Mean precision, recall, F1 score, and AUC achieved by all models of the same training and usage window sizes throughout the usage period. Like for the previous figures, we also include a variant where we only evaluate models with evaluation windows matching one of the 6-week models for better comparability.

5.4.2 Pipeline Recovery (RQ2)

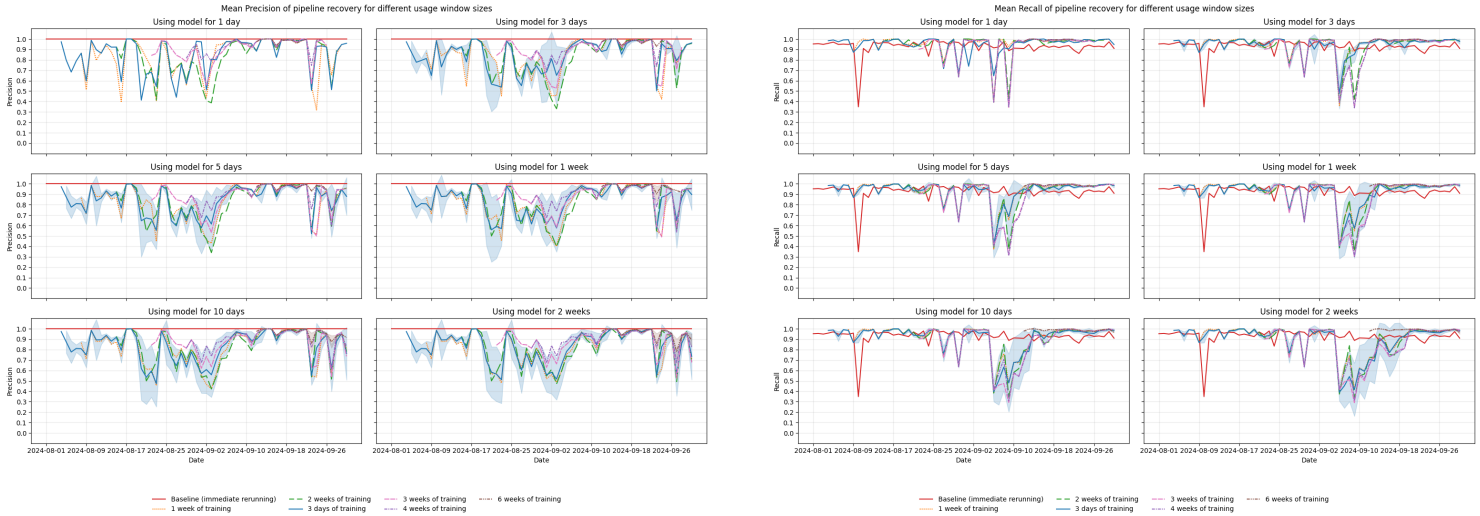
Despite the fact that our approach is unable to compete with immediate rerunning in terms of classifying flaky failures, we investigate performance regarding pipeline recovery, since this is the main practical application of our approach. The results of this analysis are shown in Figure 5.7.

On a pipeline level, our approach yields significantly better results than on a failure level. When used for just one day, our model recalls more recoverable pipelines than immediate rerunning on most days. The longer the usage window size, the worse the recall gets. TODO: wie evaluiere ich sowas wie "durchschnittliche abweichung von der baseline" über alle tage hinweg?

Focusing on the increase in flakiness in the ground truth that we observe in Figure 5.1, we see that recall immediately decreases at that time. At the end of the usage window

size, it recovers again. The rate of recovery can be explained by the fact that the curve's value for every day is the average of all the models with the same configuration that are used on that day. On days closer to the change, more models would have been in use longer and would not have seen any of this changed data in their training. For days further away from the change (with a maximum distance being imposed by the usage window size), more models would have been retrained since the change in data and would have had the chance to learn its characteristics. This finding emphasizes the need for retraining the model periodically and shows that shorter usage window sizes lead to better results in the immediate time following a change in the true flakiness distribution.

TODO: precision is SHIT



(a) Precision

(b) Recall

Figure 5.7: Precision and recall per day for different training and usage window sizes. Not that these graphs are per absolute day, not per day after training. TODO: mittig auf seite platzieren oder gar noch mehr auf die seite packen

TODO: say something about Figure 5.8

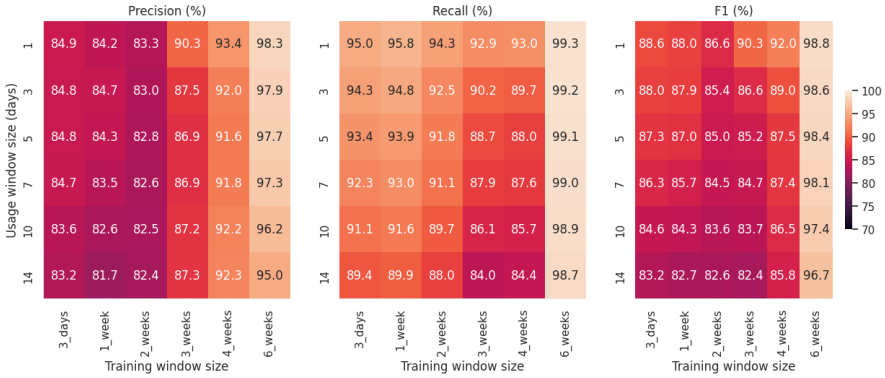


Figure 5.8: Precision, recall, and F1 score of pipeline recovery for different training and usage window sizes (mean of all models with the same window sizes).

Answer to RQ2: Our appraoch is able to correctly identify more recoverable pipelines than immediate rerunning at short usage window sizes, degrading with longer usage window sizes. However, it incorrectly recovers many non-recoverable pipelines, reaching a precision of less than 40% on some days. Precision does not suffer from longer usage window sizes.

5.5 Discussion

Our evaluation shows that readily available information in CI does indeed have predictive power for flaky failures, even when being evaluated against not just immediate reruns but also validation runs. However, the low and unreliable precision that our approach achieves is a severe threat to the practical applicability.

5.5.1 Decision Policy and Confidence

One reason for the low precision is that the current recovery rule applies a hard decision boundary without considering model confidence. A confidence-aware policy (for example, requiring higher confidence before recovering a pipeline) could potentially reduce false recoveries. While this could happen at the expense of recall, some brief investigation into the distribution of the model's prediction confidences suggests that a sweet spot exists where precision can be raised without a significant loss in recall.

5.5.2 Retraining Cadence and Distribution Shift

The temporal analyses indicate that model quality is sensitive to changes in the underlying flakiness distribution. In particular, recall decreases around distribution changes and then improves once retraining has incorporated newer data. This supports the motivation for periodic retraining from Section 4.3 and suggests that retraining cadence is a central operational parameter. A more systematic strategy for drift-aware retraining is discussed in Section 6.3.

5.5.3 Completeness of Ground Truth

The ground truth in this study depends on one immediate rerun and one validation run per failure (when available). This may not capture all flaky behavior. As a result, some model predictions that are counted as false positives could reflect true, but unobserved, flakiness. This limitation affects the interpretation of precision and should be investigated with richer validation procedures (see Section 6.2).

5.5.4 Representativeness of the Results

The results vary noticeably with the exact evaluation period, and the dataset itself exhibits strong temporal variation. This limits how confidently the observed effects can be generalized beyond the analyzed period and project. Broader evidence is needed to determine whether these patterns are specific to the selected GitLab window or typical for other systems and time spans (see Section 6.4).

5.5.5 Comparison to Other Approaches

The comparability to other approaches is highly limited by the fact that our ground truth differs from most related work as well as our approach requiring regular retraining, which is not common in related work. The inconsistency of our results further limits the comparability.

Nevertheless, comparing our results to those of Gruber et al. [14], we see that our approach yields slightly worse precision, recall, and F1 scores for all models trained on less than 6 weeks of data (ignoring those due to the limited evaluation data). However, the difference between their scores and those achieved by our best window size configuration (ignoring the 6-week models) is never more than 3.5%. The approaches can thus be considered to perform similarly in this regard (ignoring the issues with comparability mentioned above).

Hoang et al. [15] don't cite exact results achieved by their approach. They calim to achieve an AUPRC of over 90% when classifying flaky failures. With exception to 3

days of training and 2 weeks of usage, all of our configurations achieve an average AUPRC of at least 90% as well. All models perform the best for 1 day of usage, where the worst mean AUPRC score of 96.4% is achieved by the 3-day models and the best score of 98.1% is achieved by the 4-week models (ignoring the 99.9% AUPRC of the 6-week models).

On a pipeline level, Hoang et al. are able to recover over 80% of recoverable pipelines. Our approach also clears that bar, with all configurations achieving a mean recall of at least 84%. Hoang et al. report their false recovery in absolute terms (pipelines per week), which makes a comparison impossible, even aside from the issues with comparability mentioned above.

6 Future Work

While our approach yields promising results, there are still several areas that could be improved.

6.1 Confidence-Aware Recovery Decisions

6.2 Ground-Truth Completeness

6.3 Drift-Aware Retraining

6.4 Representativeness and Generalization

6.5 Feature-Space Extensions Under Practical Constraints

7 Conclusion

TODO: threats to validity: ground truth incomplete, test suites changing, model not generalizing, data not representative, the weird spikes in the data

threats to validity:

- [26] found that 95% confidence can be achieved in classification using up to 3 reruns. we do validation runs, which might have a different confidence threshold
- data leakage: features shouldn't contain any data leakage. perhaps test execution frequency could be an indication for the pipeline needing repairing (i.e. not being flaky), but the impact of such execution frequency changes should be negligible since it's not a feature of its own and the model can't learn to prefer certain branches. it can only learn that e.g. "whatever linearization produces the minimum X is the most informative".
- for gitlab, the test names are file paths. this may affect distance-based features

Abbreviations

CI Continuous Integration

SUT System under Test

TEB Test Execution Batch

VCS Version Control System

List of Figures

List of Tables

Bibliography

[1] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Jan. 31, 2022, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3476105 (cit. on pp. 1, 4, 6).

[2] M. Barboni, A. Bertolino, and G. De Angelis, "What We Talk About When We Talk About Software Test Flakiness," in *Quality of Information and Communications Technology*, A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, and R. Pérez-Castillo, Eds., vol. 1439, Cham: Springer International Publishing, 2021, pp. 29–39, ISBN: 978-3-030-85346-4 978-3-030-85347-1. DOI: 10.1007/978-3-030-85347-1_3 (cit. on p. 1).

[3] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, "Empirical Study of Restarted and Flaky Builds on Travis CI," in *Proceedings of the 17th International Conference on Mining Software Repositories*, Seoul Republic of Korea: ACM, Jun. 29, 2020, pp. 254–264, ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387460 (cit. on pp. 1, 6).

[4] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of Flaky Tests in Continuous Integration: An Industrial Case Study," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Toronto, ON, Canada: IEEE, May 27, 2024, pp. 329–340, ISBN: 979-8-3503-0818-1. DOI: 10.1109/ICST60714.2024.00037 (cit. on pp. 1, 5).

[5] 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." arXiv: 2112.04919 [cs], Accessed: Jul. 1, 2025. [Online]. Available: http://arxiv.org/abs/2112.04919 pre-published (cit. on pp. 1, 5, 6).

[6] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 830–840, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338945 (cit. on pp. 1, 4, 5, 7).

[7] M. Gruber and G. Fraser. "A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It." arXiv: 2203.00483 [cs], Accessed: Jul. 2, 2025. [Online]. Available: http://arxiv.org/abs/2203.00483 pre-published (cit. on pp. 1, 5–7).

[8] 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*, Seoul South Korea: ACM, Jun. 27, 2020, pp. 1471–1482, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3381749 (cit. on p. 1).

[9] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" In *Proceedings of the 17th International Conference on Mining Software Repositories*, Seoul Republic of Korea: ACM, Jun. 29, 2020, pp. 492–502, ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387482 (cit. on pp. 1, 8, 18).

[10] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," May 22, 2021 (cit. on pp. 1, 6, 8, 13, 18, 19).

[11] 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." arXiv: 2302.10594 [cs], Accessed: Jul. 1, 2025. [Online]. Available: http://arxiv.org/abs/2302.10594 pre-published (cit. on pp. 1, 6).

[12] P. A. C. Martins, V. A. Alves, I. Lima, C. Bezerra, and I. Machado, "Exploring Tools for Flaky Test Detection, Correction, and Mitigation: A Systematic Mapping Study," in *Anais Do IX Simpósio Brasileiro de Testes de Software Sistemático e Automatizado (SAST 2024)*, Brasil: Sociedade Brasileira de Computação, Sep. 30, 2024, pp. 11–20. DOI: 10.5753/sast.2024.3700 (cit. on p. 1).

[13] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg Sweden: ACM, May 27, 2018, pp. 433–444, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180164 (cit. on pp. 1, 6, 8).

[14] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser. "Practical Flaky Test Prediction using Common Code Evolution and Test History Data." arXiv: 2302.09330 [cs], Accessed: Apr. 17, 2025. [Online]. Available: http://arxiv.org/abs/2302.09330 pre-published (cit. on pp. 2, 13, 15–19, 33).

[15] M. Hoang and A. Berding, "Presubmit Rescue: Automatically Ignoring FlakyTest Executions," in *Proceedings of the 1st International Workshop on Flaky Tests*, Lisbon Portugal: ACM, Apr. 14, 2024, pp. 1–2, ISBN: 979-8-4007-0558-8. DOI: 10.1145/3643656.3643896 (cit. on pp. 2, 15–18, 33).

[16] F. Leinen, M. Gruber, S. S. Erdogan, A. Stahlbauer, and A. Pretschner, "An empirical study of detected and undetected flaky test failures in real-world CI pipelines," Submitted to IEEE Transactions on Software Engineering (under review), 2026 (cit. on pp. 2, 5, 11, 21, 23).

[17] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug. 1996, ISSN: 00985589. DOI: 10.1109/32.536955 (cit. on p. 3).

[18] M. Fowler. "Continuous Integration," martinfowler.com, Accessed: Apr. 21, 2025. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html (cit. on pp. 3, 4).

[19] 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*, San Jose CA USA: ACM, Jul. 21, 2014, pp. 385–396, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610404 (cit. on p. 4).

[20] 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, ISBN: 978-1-4673-7532-0. DOI: 10.1109/ICSM.2015.7332456 (cit. on p. 4).

[21] 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)*, Madrid: IEEE, Sep. 2018, pp. 1–23, ISBN: 978-1-5386-8290-6. DOI: 10.1109/SCAM.2018.00009 (cit. on p. 4).

[22] 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*, Hong Kong China: ACM, Nov. 11, 2014, pp. 643–653, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635920 (cit. on p. 4).

[23] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Xi'an, China: IEEE, Apr. 2019, pp. 312–322, ISBN: 978-1-7281-1736-2. DOI: 10.1109/ICST.2019.00038 (cit. on pp. 4, 8).

[24] H. Aïdasso, F. Bordeleau, and A. Tizghadam, "On the Diagnosis of Flaky Job Failures: Understanding and Prioritizing Failure Categories," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice*

(ICSE-SEIP), Ottawa, ON, Canada: IEEE, Apr. 27, 2025, pp. 192–202, ISBN: 979-8-3315-3685-5. DOI: 10.1109/ICSE-SEIP66354.2025.00023 (cit. on p. 5).

[25] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Surveying the developer experience of flaky tests," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 253–262, ISBN: 978-1-4503-9226-6. DOI: 10.1145/3510457.3513037 (cit. on p. 5).

[26] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An Empirical Study of Flaky Tests in Python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Porto de Galinhas, Brazil: IEEE, Apr. 2021, pp. 148–158, ISBN: 978-1-7281-6836-4. DOI: 10.1109/ICST49551.2021.00026 (cit. on pp. 6, 36).

[27] A. Alshammari, P. Ammann, M. Hilton, and J. Bell, "230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Toronto, ON, Canada: IEEE, May 27, 2024, pp. 257–268, ISBN: 979-8-3503-0818-1. DOI: 10.1109/ICST60714.2024.00031 (cit. on p. 8).

[28] V. Pontillo, F. Palomba, and F. Ferrucci, "Static test flakiness prediction: How Far Can We Go?" *Empirical Software Engineering*, vol. 27, no. 7, p. 187, Dec. 2022, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-022-10227-1 (cit. on p. 8).

[29] W. Ge and C. Zhang. "Understanding and Detecting Flaky Builds in GitHub Actions." arXiv: 2602.02307 [cs], Accessed: Feb. 26, 2026. [Online]. Available: http://arxiv.org/abs/2602.02307 pre-published (cit. on p. 8).

[30] F. Moriconi, R. Troncy, A. Francillon, and J. Zouaoui, "Automated Identification of Flaky Builds using Knowledge Graphs," in *Proceedings of the 23rd International Conference on Knowledge Engineering and Knowledge Management*, Bozen-Bolzano, Italy, Sep. 2022 (cit. on p. 12).

[31] 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. 20, 2021, pp. 1381–1392, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3473931 (cit. on p. 17).

[32] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know You Neighbor: Fast Static Prediction of Test Flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3082424 (cit. on p. 18).

[33] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage AK USA: ACM, Jul. 25, 2019, pp. 2623–2631, ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330701 (cit. on p. 19).