FlaDaGe: A Framework for Generation of Synthetic Data to Compare Flakiness Scores

Mert Ege Can[1], Joanna Kisaakye[1,2], Mutlu Beyazıt[1,2] and Serge Demeyer[1,2]

[1]Universiteit Antwerpen, Belgium
[2]Flanders Make vzw, Belgium

Abstract

Several industrial experience reports indicate that modern build pipelines suffer from flaky tests: tests with non-deterministic results which disrupt the CI workflow. One way to mitigate this problem is by introducing a flakiness score, a numerical value calculated from previous test runs indicating the non-deterministic behaviour of a given test case over time. Different flakiness scores have been proposed in the white and grey literature; each has been evaluated against datasets that are not publicly accessible. As such, it is impossible to compare the different flakiness scores and their behavior under different scenarios. To alleviate this problem, we propose a parameterized artificial dataset generation framework (FlaDaGe), which is tunable for different situations, and show how it can be used to compare the performance of two separate scoring formulae.

Keywords

Flakiness, Flakiness scores, Continuous Integration, Automation

1. Introduction

Software testing is a vital necessity for modern software engineering, ensuring system reliability, quality, and developer productivity in environments of increasing complexity [1, 2]. In continuous integration (CI) pipelines, automated test suites are executed to validate software works as intended before every deployment. However, these pipelines face a critical challenge: flaky tests [3, 4].

Flaky tests are tests with non-deterministic outcomes, switching between different results under identical conditions. This behavior reduces trust in the test results, wastes engineering efforts, and disrupts CI workflows [5, 6]. Large-scale industrial studies within firms such as Google, SAP and Microsoft show that flakiness continues to appear across projects and progressively worsens over time, causing significant costs in debugging and pipeline stability [7, 8].

Despite the increasing attention to flaky tests from the research community, there is still one principal challenge test engineers must grapple with, pertaining to the selection of a mitigation strategy. Before a mitigation strategy may be selected, the magnitude of the flakiness problem must be assessed. This is achieved through the use of flakiness scores and several scores exist in the white and grey literature [9, 10, 11, 12, 13, 14, 15]. However, most of the scores presented in the literature are evaluated against a dataset that is not accessible to the public. Indeed, at the time of this writing, there is no benchmark dataset against which multiple flakiness scoring techniques can be assessed.

This lack of standardized assessment frameworks allowing for a direct comparison of different flakiness scoring algorithms makes it harder to understand the strengths and weaknesses of the algorithms. Existing approaches are often evaluated in isolated configurations with unique use cases, making it difficult to establish a common baseline between algorithms. Moreover, a number of current research reports are based on datasets derived from real-world systems or undisclosed industrial case studies, which are individually valuable but do not always provide controlled conditions for systematic evalua-

The 24th Belgium-Netherlands Software Evolution Workshop (BENEVOL 2025)
m.egecan@gmail.com (Mert Ege Can); joanna.kisaakye@uantwerpen.be (Joanna Kisaakye); mutlu.beyazit@uantwerpen.be (Mutlu Beyazıt); serge.demeyer@uantwerpen.be (Serge Demeyer)
0009-0006-2595-0311 (Mert Ege Can); 0000-0001-7081-5385 (Joanna Kisaakye); 0000-0003-2714-8155 (Mutlu Beyazıt); 0000-0002-4463-2945 (Serge Demeyer)

tion. This gap hinders the ability to fairly assess the performance of competing algorithms in varying flakiness scenarios.

To address these challenges, this research introduces a configurable dataset generation framework (FlaDaGe) designed to simulate test suite behaviors with controllable flakiness trends. The framework is algorithm-neutral, which means that it is not biased towards any algorithm and allows for the systematic creation of artificial datasets with varying flakiness patterns. Finally, we demonstrate how it can be used to create a simulated dataset to compare two flakiness scoring algorithms; the No-Fault-Found (NFF) rate [11] and the Extended Flakiness Score (EFS) [12].

2. Flakiness Scores

Although re-running, monitoring, and fixing provide combined response strategies to flakiness, they all benefit from a unified quantification to rank which tests are most in need of attention. Flakiness scoring serves this purpose by calculating a numerical value for each test representing its level of instability [9, 10, 11, 12, 13]. The flakiness score of a test is derived from analyzing its execution history over a defined period, such as several days, weeks, or months in the CI pipeline. This score represents the degree of inconsistency in the results of a test, based on the ratio and frequency of change between the different test results. Tests that consistently produce the same result are considered stable, whereas alternating results are marked as more flaky.

In practice, a test with a high flakiness score means its results are unpredictable, and thus a candidate for re-run validation, long-term monitoring, or root cause analysis and fixing. For example, suppose that two different tests each fail three times in the last 20 runs. Test A fails in the first three consecutive runs, while test B fails sporadically such that failures occur in the third, eighth, and fourteenth runs. Although their failure counts are the same, Test B will exhibit more complexity, an unpredictable pattern when it fails, and therefore a higher flakiness score, reflecting its greater non-deterministic nature.

Flakiness scoring introduces several concrete benefits throughout the life cycle of software testing.

1. **Prioritization**: By being able to rank tests according to their instability, the development effort can be focused on the improvements with the greatest impact on the reliability of the system.

2. **Visualization**: Scoring enables graphical representations of flakiness statistics. Heat maps or historical flakiness plots offer valuable insight into the characteristics of flaky tests.

3. **Alerting**: Automated systems can flag tests whose flakiness score exceeds a predefined instability threshold, suggesting a review by the engineer.

4. **Tracking**: Flakiness scores can be tracked over time to assess performance in system improvements or component degradation.

Flakiness scoring also complements the re-run, monitor, and fix cycle. During re-runs, scoring helps decide whether additional executions are needed to reach a confident conclusion. In monitoring, scores support detecting trends depending on the change in score such as, constantly increasing values would mean instability is increasing as time advances. In the fix phase, the flakiness score can enable assessing the condition before and after the solution, validating whether the applied fix has effectively reduced the flakiness. Ultimately, flakiness scoring introduces an automated means of managing test instability in CI environments, allowing automated flaky test identification. In doing so, it strengthens the overall reliability and maintenance of the testing suite, ensuring that the effort spent is efficient and carefully maintained throughout the software delivery pipeline.

Table 1 shows a summary of the flakiness score definitions defined in the white and grey literature at the time of this writing. As shown in the table, the majority of flakiness scores proposed in the literature so far rely solely on the presence of test result history justifying the creation of an artificial dataset generation framework with which to generate data to compare different scoring algorithms.

Table 1
Comparison of flakiness score formulae and the data required to compute the score.

#	Paper (Year)	Formula	Required Data Elements
1	Kowalczyk et al.,(2020) [9]	Entropy: $f(R_{v,*}) = -\sum_{i\in(P,F)} p(i)\log_2 p(i)$ Flip rate: $f(R_{v,*}) = \frac{numFlips(R_{v,*})}{numPossibleFlips(R_{v,*})}$ Unweighted Average: $U_H(R) = \sum_{v\in V^t_{t-H}} \frac{f(R_{v,*})}{\lvert V^t_{t-H}\rvert}$ Weighted Average: $W_{\lambda,P}(R,n) = Z_n = \frac{\lambda x_n + (1-\lambda)Z_{n-1}}{\lambda\sum_{i=0}^{n-1}(1-\lambda)^i}$ where $x_n = \sum_{v\in V^{nP}_{(n-1)P}} f(R_{v,*})$	Test Result History. Weights.
2	Gruber et al.,(2023) [10]	Flip rate: $flip\_rate(R) =$ $\sum_{t=1}^{n-1}\left(\frac{1}{n-1}\cdot\begin{cases}1, & \text{if } r_t \neq r_{t+1}\\ 0, & \text{if } r_t = r_{t+1}\end{cases}\right)$ Decayed Flip rate: $flip\_rate(R,w) =$ $\sum_{t=1}^{n-1}\left(\frac{w(t)}{\sum_{u=1}^{n-1} w(u)}\cdot\begin{cases}1, & \text{if } r_t \neq r_{t+1}\\ 0, & \text{if } r_t = r_{t+1}\end{cases}\right)$	Test Result History. Weight functions and Weights.
3	Rehman et al.,(2021) [11]	NFF rate: $\text{NFFRate}_t = \frac{f}{r}$ Stable NFF rate: $\text{StableNFFRate}_t = \text{NFFRate}_t(f,r)$ Likelihood: $P_t(f,r,p) = \binom{r}{f}\cdot p^f\cdot(1-p)^{r-f}$ where $p = \text{StableNFFRate}_t$	Test Result History. Test Report History (Whether an issue or bug was filed for the test).
4	Kisaakye et al., (2024) [12]	Transition rate: $T(R_{v,*,\{r_1,r_2\}}) = \frac{numTransitions(R_{v,*,\{r_1,r_2\}})}{numTotalTransitions(R_{v,*})}$ Multi transition rate: $\sum_{\{i,j\}\subseteq\{r_1\ldots r_n\},i\neq j} T(R_{v,*,\{i,j\}})$ Unweighted Average: $U_H(R) = \sum_{v\in V^t_{t-H}} \frac{f(R_{v,*})}{\lvert V^t_{t-H}\rvert}$ Weighted Average: $W_{\lambda,P}(R,n) = Z_n = \frac{Y_n}{\lambda\sum_{i=0}^{n-1}(1-\lambda)^i}$ where $x_n = \sum_{v\in V^{nP}_{(n-1)P}} \frac{f(R_{v,*})}{\lvert V^{nP}_{(n-1)P}\rvert}$ and $Y_n = \lambda x_n + (1-\lambda)Y_{n-1}$	Test Result History. Weights.
5	Facebook/Meta Eng. Blog, (2020) [13]	Probabilistic Flakiness Score: $\text{PFS} = P(\text{Failure} \mid \text{good state})$ (estimated via Bayesian model).	Test Result History. context/features to distinguish "good" vs "bad" state Bayesian priors for model parameters.
6	Rasheed et al., (2020) [14]	Flakiness Score: The variability between test runs.	Test Result History.
7	Haben et al., (2024) [15]	Flake rate: $flakeRate(t,n) =$ $\frac{1}{w}\sum_{x=n-w}^{n-1} flake(t,x)$ Where $flake(t,x)$ is defined as: $flake(t,x) =$ $\begin{cases}1, & \text{if test } t \text{ flaked in build } b_x\\ 0, & \text{otherwise}\end{cases}$	Test Result History.

3. Constructing a Fair Evaluation Ground

This study has one driving research question.

How can we design a unified and statistically controlled dataset that enables a fair and algorithm-neutral comparison of different flakiness scoring algorithms?

To compare two fundamentally different scoring models, a shared evaluation ground is required. This dataset should not encode the assumptions or internal mechanics of any algorithm. In response to this, we propose a framework to enable the generation of an artificial dataset designed to simulate flakiness patterns in a controlled and observable way. Test executions are modeled across multiple versions, and result sequences are generated using probabilistic functions that reflect common flakiness patterns.

The dataset should also support the input requirements of all currently available algorithms. By examining Table 1, we can see that most flakiness algorithms require a test result history and one a test report history. For this purpose, we select one algorithm as a representative for all algorithms that require a test result history, the one proposed by Kisaakye et al. [12], and another representative of those algorithms that require test report presence, Rehman et al. [16]. By studying the requirements of these two algorithms, we are able to gather the broad spectrum of requirements for an artificial dataset generation framework.

4. Requirements for a Dataset

Before compiling the requirements for a dataset generation framework, it is essential to understand the characteristics of the datasets originally used in the evaluation of the representative flakiness scoring algorithms.

- Rehman et al. evaluated the NFF algorithm using industrial-scale test execution data collected from Ericsson's CI environment [16]. Crucially, each test run is labeled with one of the two possible binary outcomes, Pass and Fail, and information about the presence of fault reports, aligning with the definition of flakiness used by the NFF algorithm, failures without a report.
- Kisaakye et al. evaluated the EFS algorithm against artificial datasets designed to emulate flakiness using controlled statistical parameters [12]. The first is created using the dataset generation algorithm proposed by Kowalczyk et al., and models flakiness using probability mass functions in which case each test case has a static flakiness probability, and each version has a probability of revealing a fault [9]. The second adds support for more test results, Error and Skip, and more complex flakiness trends such as Increasing, Decreasing, Sporadic, Sudden spikes or drops.

Therefore, to allow a valid comparison between such models, a "good" dataset must meet these essential criteria:

- **Different Result States:** The dataset should include different result states to model real world build systems that have more results than Pass and Fail.
- **Report Association:** Some failure states must include report flags, and others must lack them, to simulate realistic distributions of fault attribution. This is a necessary requirement for all flakiness scoring algorithms that define flakiness as a failure in the absence of a test report.
- **Varied Flakiness Trends:** Flakiness should evolve over time using various trend profiles, such as increasing, decreasing, or stable, to test the ability of each algorithm to react to environmental variations.
- **Version and Run Structure:** The dataset must have a clear definition of version and run to allow scoring models to assess the historical progression of the test behavior. Each test should have multiple runs per version, for multiple versions.

5. Artificial Dataset Generation Framework

The dataset generation framework is built around six concepts: *Test*, *Suite*, *Version*, *Run*, *Trends*, and *Report Presence*. These concepts are often represented intuitively in the literature, however, they require clear definitions since they are necessary for most of the flakiness scores presented in Table 1. They aid in modeling different dimensions of flakiness behavior, and, when systematically combined, simulate evolving test flakiness.

5.1. Test

A *Test* represents the fundamental unit of analysis in the dataset. It encapsulates flakiness flags, flakiness probability, flakiness probability change (δ), and a population of versions.

Within this framework, each test is represented as a simulated execution trace spanning multiple versions each with hundreds of individual runs. Each test is assigned a fixed flakiness flag at the beginning of the dataset generation process: *clear*, *faulty* or none which means the test is *flaky*. *flaky* tests are eligible to receive flaky results, while *clear* and *faulty* tests are constrained to emit only Pass or Fail results, respectively. The probability of flakiness for *flaky* tests is randomly assigned based on predefined thresholds. This flakiness probability is forwarded to versions where the effect of the version trend is calculated, and determines the occurrence of a flaky result for every run generated within the version. The distinction between flaky and non-flaky tests ensures that the ground truth flakiness of each test is known in advance, forming the basis for evaluating how accurately an algorithm can differentiate this status based solely on test results.

5.2. Suite

A *Suite* is the highest-level organizational unit in the dataset generation framework. It encapsulates an entire simulation scenario defined by a unique combination of flakiness trends and execution parameters. Each suite includes a population of tests and a consistent configuration of version and run trends, thereby modeling a testing environment.

5.3. Version

A *version* in the framework represents a high level temporal segment within the dataset, simulating a specific period in the development cycle. The *version* is equivalent to a unique state of the software product within a CI system, capturing a specific execution configuration of the whole ecosystem, and acting as a reproducible environment for automated test execution. It can be represented by various identifiers depending on the development or deployment environment used, such as a build number or a tagged release.

The first version typically serves as a control baseline, while subsequent versions re-calculate the version-level flakiness probability according to version trend. The change on top of the test-level flakiness probability depends on the selected version trend, and is calculated using the δ value and the ratio of current version to total version count. By modeling multiple versions, the dataset generated reflects the evolving nature of development workflows.

5.4. Run

A *run* within the framework is the smallest unit of the dataset that carries the core information of each individual test execution performed for a test in each version. This concept corresponds to a run within a CI system which is a single execution of a test within a particular version. When a *run* is generated, the result is decided independently according to a pre-defined flakiness probability assigned to the test for the current version.

Run-level flakiness probability, is calculated in a manner similar to version-level flakiness probabilities according to the selected run trend of the suite. At this level, the run scales the version-level flakiness probability using the ratio of the current run to the total runs.

5.5. Trends

Once the test results are organized by version and run, it is possible to interpret and characterize emergent flakiness behaviors as *trends*. Trends provide time-wise information about test stability. Recognizing these trends allows test engineers to correlate the evolution of flakiness and take smarter precautions or apply more targeted solutions.

Flakiness trends can be analyzed at two levels; across *runs*, i.e, **run trend**, how individual test results evolve over multiple executions within the same version, and across *versions*, i.e., **version trend**, how the overall flakiness score for a given test changes from one version to the next. Table 2 summarises the trends implemented within the framework and how they affect flakiness behavior at the version and run level. The trends presented in Table 2 build upon those presented in [12] and aim to generalise the flakiness patterns found in practice. For example, the application of a direct fix for a single test within a single version, such as a code change, should trigger a sudden drop in test flakiness, while the application of an indirect fix, such as network stabilization, may only be observed as gradually decreasing flakiness.

5.5.1. Version Trend

The trend of the version defines the evolution of the flakiness by specifying a δ value and changing the base flakiness of each test by that amount accordingly.

These patterns are configured globally per suite and applied across all flaky tests in that suite. This ensures statistical consistency while maintaining a controlled environment for evaluating the performance of the algorithms. The dataset generation process implements the trends in Table 2 as three types of change patterns: *linear*, *exponential*, and *sudden*, which control how the δ value is applied to intermediate versions:

$$\text{Linear:} \quad p_v = p_t \pm \delta \cdot \frac{v}{V} \tag{1}$$

$$\text{Exponential:} \quad p_v = p_t \pm \delta \cdot \left(\frac{v}{V}\right)^2 \tag{2}$$

$$\text{Sudden:} \quad p_v = \begin{cases} p_t, & v < T \\ p_t \pm \delta, & v \geq T \end{cases} \tag{3}$$

where p_v is the version-level flakiness probability v, p_t is the test-level flakiness probability, V and v the total number and the current number of versions, respectively. T is the threshold version and, when reached, the δ value is applied in full. The positive and negative signs in the formulae depend on whether the trend is *increasing* (positive) or *decreasing* (negative).

As an example, if a test is assigned a test-level flakiness probability of 0.3 and the δ value is set to 0.15, then:

- For a decreasing trend, the probability that the final version is reached would be $0.3 - 0.15 = 0.15$.
- For an increasing trend, the probability that the final version is reached would be $0.3 + 0.15 = 0.45$.
- For a uniform trend, the probability would remain constant at 0.3.

Table 2
Trends implemented within the framework and the induced behavior at the Version and Run level

Trend	Version	Run
Uniform	Every flaky test retains its initial flakiness probability.	The flakiness probability does not change within a version. Flaky instances are spread randomly across the run sequence.
Increasing	The initial flakiness probability of every flaky test increases linearly across versions.	The flakiness probability starts from the probability 0 and linearly increases to the version level flakiness probability. Flakiness is concentrated in the later runs.
Decreasing	The initial flakiness probability of every flaky test decreases linearly across versions.	The flakiness probability starts from the version-level flakiness probability and linearly decreases down to 0 probability. Flakiness is concentrated in earlier runs.
Exponentially Increasing	The initial flakiness probability of every flaky test increases exponentially across versions.	The flakiness probability starts from the probability 0 and increases exponentially up to the version-level flakiness probability. Flakiness is concentrated in the later runs, with an accelerating increase.
Exponentially Decreasing	The initial flakiness probability of every flaky test decreases exponentially across versions.	The flakiness probability starts from the version-level flakiness probability and decreases exponentially to 0 probability. Flakiness is concentrated in earlier runs, with an accelerating decrease.
Suddenly Increasing	The initial flakiness probability of every flaky test increases by the δ value after a specific version is reached.	All flakiness occurs in a window after a specific run is passed. The flakiness probabilities are set to the probability 0 before the specific run and then the version-level flakiness probability is assigned.
Suddenly Decreasing	The initial flakiness probability of every flaky test decreases by the δ value after a specific version is reached.	All flakiness occurs in a window before a specific run is passed. The flakiness probabilities are set to the version-level flakiness probability before the specific run, and then the 0 probability is assigned.

5.5.2. Run Trend

While the version trend controls the overall flakiness per version, the run trends in Table 2 determine their temporal distribution within a version. This trend simulates realistic scenarios where test flakiness may not be evenly distributed across the execution timeline.

Decoupling the run trend from the version trend allows independent control over the frequency and time when failures occur. This separation is essential to evaluate the ability of each algorithm to detect both distributed and localized flakiness, revealing the strengths and weaknesses of each algorithm in varying testing scenarios.

Unlike version-level, where the δ value represents the difference between the probabilities of the first and last versions, the run trend scales the run-level flakiness probability between 0 and p_v. This means that the probability at a given run p_r is calculated as:

$$p_r = p_v \cdot \frac{r}{R}$$

where the scaling ratio is derived from R and r, the total run count and the current run number, respectively. The dataset implements the same change patterns: *linear*, *exponential*, and *sudden*. The run-level flakiness probabilities are calculated as:

$$\text{Linear:} \quad p_r = p_v \cdot \pm \frac{r}{R} \tag{4}$$

$$\text{Exponential:} \quad p_r = p_v \cdot \pm \left(\frac{r}{R}\right)^2 \tag{5}$$

$$\text{Sudden:} \quad p_r = \begin{cases} 0, & \frac{r}{R} < \rho_T \\ p_v, & \frac{r}{R} \geq \rho_T \end{cases} \tag{6}$$

where ρ_T is the threshold ratio such that $\rho_T = 0.5$ corresponds to the halfway point in the run sequence.

5.6. Report Presence

Although flakiness scoring provides quantitative values for identifying flaky tests, it does not naturally differentiate between explainable and unexplainable failures. In practical terms, not all failed tests with a high failure count or sporadic occurrences present the same flakiness severity. The availability of an attached failure report represents a documented explanation of why a test failed, which may include information on the root cause, or context possibly reducing the effort required to address the problem and ultimately the severity of flakiness.

Within the framework, this report presence is represented by a report flag attached to each test. When a test is generated, the report flag is also decided independently according to a specified probability.

6. Results

We generate and evaluate a dataset using the framework, available in our replication package [17]. During evaluation, we focus on dataset creation and the extent to which the generated dataset captures the characteristics necessary for a meaningful assessment of the two exemplary scoring algorithms, the No Fault Found (NFF) algorithm by Rehman et al. [16] and the Extended Flakiness Score (EFS) by Kisaakye et al. [12]. This includes demonstrating how the dataset meets the established criteria for a "good" dataset, by examining the distribution of result and report associations, the diversity of flakiness patterns, and how the two algorithms would "observe" the dataset. These characteristics are demonstrated using graphs of version trends and run trends, as well as visual representations of flakiness probabilities of the ground truth of a single suite. This suite selected is the one with the trend pair of increasing version flakiness and decreasing run flakiness, which will be called the **increase-decrease suite** for simplicity throughout the rest of this section. The example suite is one of the 49 generated suites within the dataset, chosen because it tries to portray a realistic development scenario: A situation in which overall flakiness increases with each version, due to the growing complexity of the software, meanwhile the development team continuously works on improving the system stability and fixing flaky tests during a version so the run-level flakiness gradually decreases.

6.1. Dataset Overview

By iterating through all the combinations of the version and run trends discussed in Section 5, the framework creates a dataset with 49 unique test suites, each modeling a distinct flakiness pattern simulated over a year.

Each suite consists of; **100 tests** each simulated with **4 versions** with **250 runs** in each version. This yields a total of **5,000,000 result entries** for investigation.

Each test run in the dataset is represented by the following fields, ensuring compatibility with different flakiness scoring models:

- `Test ID`, `Release ID`, and `Run ID` to support detailed tracking of each test result.
- `Report Flag` indicating whether a report was created for the test.
- `Verdict` One of (`Successful`, `Fail`, `Error`, `Skip`) indicating the actual result.
- `Execution Timestamp`, representing the time at which the test was executed.

6.1.1. Test-Level Flakiness Assignment

Figure 1 presents the average base flakiness probability assigned to each test in the selected increase–decrease suite, sorted in ascending order. This base probability is not the final flakiness, but rather serves as an initial parameter from which the flakiness for each version will be derived according to the assigned version trend.

In this configuration, the flakiness probabilities assigned at the test level for flaky tests range between 10% and 40%. The distribution of test categories is 20% clear tests, 20% faulty tests, and 60% flaky tests, reflecting a scenario in which the majority of the test suite is flaky to varying degrees.

6.1.2. Version-Level Flakiness Assignment

Table 3 describes, and Figure 2 illustrates each of the seven version trends by their version-level flakiness probability averages.

Table 3
Version-Level Flakiness Trend Types

Trend	Summary	Equation	Figure
Uniform	Flakiness remains constant across versions.	—	2g
Decrease	Flakiness changes linearly across versions *with a negative slope*.	1	2a
Increase	Flakiness changes linearly across versions *with a positive slope*.	1	2c
Decrease Exponential	Flakiness changes exponentially across versions *with a negative rate* (slow early decrease, faster later).	2	2b
Increase Exponential	Flakiness changes exponentially across versions *with a positive rate* (slow early increase, faster later).	2	2d
Sudden Decrease	Flakiness changes suddenly at threshold version T *by a negative step*.	3	2e
Sudden Increase	Flakiness changes suddenly at threshold version T *by a positive step*.	3	2f

6.1.3. Run-Level Flakiness Assignment

The Table 4 describes and Figure 3 illustrates each of the seven run trends by their run-level flakiness probability averages.

Table 4
Run-Level Flakiness Trend Types

Trend	Summary	Equation	Figure
Uniform	Flakiness remains constant across runs.	—	3g
Decrease	Flakiness changes linearly across runs *with a negative slope*.	4	3a
Increase	Flakiness changes linearly across runs *with a positive slope*.	4	3c
Decrease Exponential	Flakiness changes exponentially across runs *with a negative rate* (slow early decrease, faster later).	5	3b
Increase Exponential	Flakiness changes exponentially across runs *with a positive rate* (slow early increase, faster later).	5	3d
Sudden Decrease	Flakiness changes abruptly at threshold run T_r *by a negative step*.	6	3e
Sudden Increase	Flakiness changes abruptly at threshold run T_r *by a positive step*.	6	3f

6.2. Assumptions on Flakiness Distribution

To ensure a fair, yet challenging, evaluation setting, the probabilities were carefully chosen. 20% are designated as *clear*, always passing, and another 20% as *faulty*, always failing. This guaranties that the presence of non-flaky tests, existing in equal proportions within the dataset, providing an equal baseline by which flakiness can be distinguished. The remaining 60% of the tests are *flaky*. Each *flaky* test was assigned a base flakiness probability drawn uniformly from the range of 0.1 to 0.4. This uniform assignment ensures diversity in flakiness severity while avoiding bias toward particular instability levels.

When generating each individual run, if the execution was *flaky*, its outcome was randomly determined according to additional probability weights. Reports were attached in only 20% of such *flaky* runs, simulating the noise of a system. *flaky* results were assigned: skip (10%), error (10%), and fail (80%). This selection ensures failures are the dominant, while still providing a variety of result states.

This probability design was motivated by two goals:

- Setting 60% of the dataset as *flaky*, slightly more than half of all tests, provides sufficient coverage for evaluation while preserving a significant share of deterministic results.
- The distribution of the presence of the report (20%) was aligned with the combined proportion of skip and error results (10% + 10%). In this way, both algorithms face an equal share of *non-fail* result states.

6.3. Algorithm Perspectives

In this section, we examine how the NFF and EFS algorithms interpret the artificially generated dataset, focusing on the increase-decrease suite as the representative example.

6.3.1. Trend Correlation with Ground Truth

Figure 4a and Figure 4c present the average NFF Rate computed in the example suite, allowing a direct comparison with the ground truth trends shown in Figure 4b and Figure 4d. The version-level analysis in Figure 4a shows an increasing average of the NFF Rate over successive versions. This pattern is consistent with the expectation from ground truth as the system evolves and its complexity increases. The run-level analysis in Figure 4c reveals a decreasing trend in the average NFF rates on run basis. This reflects the ongoing stabilization efforts during the version, aligning with the ground truth towards the later runs.

The alignment between these NFF Rate patterns and the ground truth demonstrates that the intended version and run trends are preserved in NFF algorithm-specific metrics. This outcome confirms that the NFF algorithm successfully captures the underlying flakiness dynamics through its own metric definitions.

6.3.2. Outcome Correlation of NFF and EFS

Figure 5a and Figure 5b present the outcome ratios according to the assumptions embedded within the NFF and EFS algorithms. When these figures are compared, both algorithms are observed to capture the intended outcome composition of approximately 20% *clear*, 20% *faulty*, and 60% *flaky* tests.

The perspective of the EFS algorithm decomposes all possible outcomes *successful*, *skip*, *error*, and *fail*. The NFF algorithm does not differentiate between *clear* and *faulty* tests; all tests rather than failures without a report are visualized in green.

When comparing the two plots, a strong correlation is observed in the maximum and minimum ratios of flaky results. Both algorithms consider the flakiest tests at around 20% flaky result ratio across all runs, with the remaining flaky tests showing a gradual decrease over the suite.

The slight difference in the ratio between the NFF and EFS plots originates from the recognition of skip (yellow), error (purple) states, and faulty tests (full red) by the EFS algorithm that are not considered

individually by the NFF algorithm. In addition to this visual difference, both graphs correlate with each other in outcome variance and distribution in terms of providing a fair comparison ground for both algorithms.

6.3.3. History Correlation of NFF and EFS

Figure 6a and Figure 6b present the test execution results for the first 20 tests of the last version of the example suite. These histories visualize how each algorithm observes run-level flakiness according to its mathematical definition of flakiness. NFF looks at report presence, while EFS looks at test outcomes directly. Despite the definitional differences of flakiness, both algorithms exhibit strong alignment in the positional distribution of flaky outcomes such that the same runs are generally marked as flaky in both histories. This alignment demonstrates that the dataset satisfies run-level compatibility for algorithm-agnostic comparisons.

Another notable observation is the visible influence of the decreasing run trend on the positioning of the flaky outcomes. It can be observed that flaky outcomes are concentrated at the start of the version. This pattern is consistent across both algorithms, further validating that the simulated dataset accurately embeds the intended run trend characteristics.

6.4. Summary of Dataset Generation Results

The results presented in this section demonstrate that the artificially generated dataset successfully embeds the intended version and runs trends while maintaining algorithm-natural compatibility. By configuring diverse combinations with version-level and run-level flakiness probability distributions, the dataset captures a wide range of possible software testing scenarios. The ground truth plots confirm these patterns from the perspective of the target algorithms. The functional requirements of each algorithm are satisfied by the generated dataset on report flags, result variety, and varying distributions of flakiness.

Furthermore, algorithm-specific analyses for NFF and EFS show that both algorithms are able to detect trends within the underlying dataset configuration, despite differences in their definitions of flakiness. The close correlation between algorithm-specific metrics and definitions validates the integrity of the dataset and ensures that the comparative analysis between NFF and EFS can be performed on a fair and representative basis.

7. Related Work

The International Dataset of Flaky Tests (IDoFT) is a collection of flaky tests in Java and Python represented as project URLs along with identifying features such as the commit when flakiness was detected, module path, fully qualified test name, category, and status [18]. While this dataset is invaluable to test flakiness research, the absence of actual test results hinders it's usability as a benchmark for the comparison of different flakiness scoring algorithms.

To address this gap, Wendler and Winter published a regression test history dataset to aid test flakiness research [19]. Their dataset features eleven module-flakiness introducing commit combinations from 8 Maven projects included within the IDoFT dataset. The complete dataset contains 28200 test result histories for 840 tests with history lengths ranging from 1 to 474 commits. This dataset would be an excellent first step toward comparing flakiness scoring algorithms against real data. However, since the underlying probability of flakiness is unknown, it cannot enable a rigorous comparison of flakiness scoring formulae and algorithms under different situations. This is the gap our work seeks to fill.

Regarding dataset generation, one other study comes close to our work. FLAKYRANK is a ranking framework proposed by Wang et al. that relies on augmented learning principles [20]. To address the under-representation of flaky tests in their training dataset, Wang et al. used Generative Adversarial Networks to create synthetic examples. However, the purpose of their dataset generation is different from our work. Their dataset is based on the FlakeFlagger dataset [21]. Therefore, their dataset

generation approach is designed to create a dataset to evaluate approaches that identify flakiness without rerunning. As such, test result and report history are not part of the features in their dataset generation approach.

8. Threats to Validity

One threat concerns the gap between the flakiness of the artificial and real world. In our dataset generation, we proposed a generic way of the data creation process by combining pre-defined trends. Although this approach provides a systematic way to simulate controlled test histories, real-world flakiness features occurring in unique patterns or edge cases may not be captured by our method.

In addition, since we re-implemented the flakiness scoring algorithms, our interpretation may differ from the one used in the original works. However, we do not expect a considerable deviation from the original implementations because we used the same formulae presented in their works.

Finally, the choice of algorithms we implemented for this work, while representative, presents a threat since we do not capture any edge cases that may occur during scoring. However, we do not expect large deviations in the way algorithms relying solely on test execution history will observe the generated dataset.

9. Conclusion

Recognising the role of flakiness scores as decision support during the process of flakiness mitigation, this work examined the different flakiness scoring algorithms proposed in the white and grey literature. We identified a gap in the literature pertaining to a mechanism through which to generate reproducible datasets to rigorously assess the strengths and weakness of different flakiness scoring algorithms. To address this gap, we developed an algorithm-neutral dataset generation framework (FlaDaGe) that can be used to model different flakiness situations and assess different flakiness scoring algorithms. Finally, we demonstrated how it can be used to generate a dataset to assess the performance of two pre-existing flakiness scoring algorithms defined in [16] and [12], showing how each algorithm "observes" the simulated flakiness. In future work, the dataset generation framework can be extended to support a wider range of features necessary for new flakiness scoring algorithms. A broader evaluation against the rest of the algortihms, and a comparison of their performance against real world data, may also be conducted.

Acknowledgments

This work is supported by the Research Foundation Flanders (FWO) via the BaseCamp Zero Project under Grant number S000323N.

References

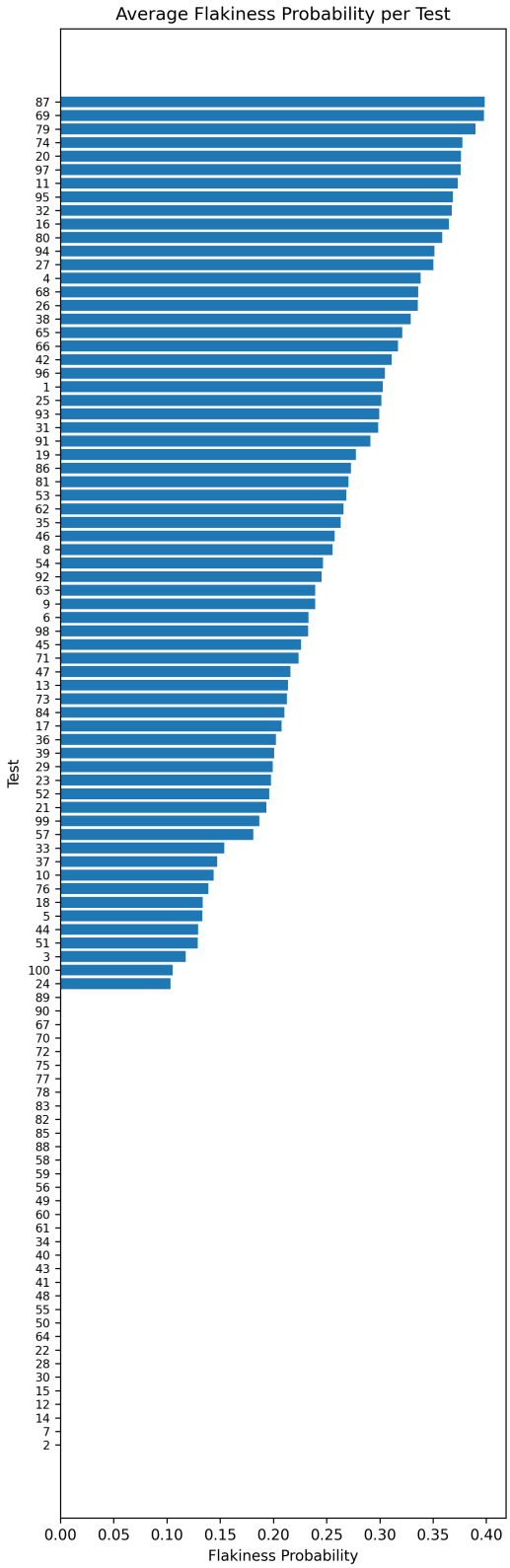
[1] F. Lonetti, E. Marchetti, Emerging software testing technologies, in: Advances in computers, volume 108, Elsevier, 2018, pp. 91–143.
[2] A. Bertolino, Software testing, SWEBOK 69 (2001).
[3] O. Parry, G. M. Kapfhammer, M. Hilton, P. McMinn, A survey of flaky tests, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (2021) 1–74.
[4] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, L. Zhang, Test flakiness' causes, detection, impact and responses: A multivocal review, Journal of Systems and Software 206 (2023) 111837.
[5] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, in: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 2014, pp. 643–653.

[6] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, J. Bell, A large-scale longitudinal study of flaky tests, Proceedings of the ACM on Programming Languages 4 (2020) 1–29.
[7] W. Lam, K. Muşlu, H. Sajnani, S. Thummalapenta, A study on the lifecycle of flaky tests, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1471–1482.
[8] A. Berndt, S. Baltes, T. Bach, Taming timeout flakiness: An empirical study of sap hana, in: Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, 2024, pp. 69–80.
[9] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, 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, 2020, pp. 110–119.
[10] M. Gruber, M. Heine, N. Oster, M. Philippsen, G. Fraser, Practical Flaky Test Prediction using Common Code Evolution and Test History Data , in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 210–221. URL: https://doi.ieeecomputersociety.org/10.1109/ICST57152.2023.00028. doi:10.1109/ICST57152.2023.00028.
[11] M. H. U. Rehman, P. C. Rigby, Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1371–1380.
[12] J. Kisaakye, M. Beyazıt, S. Demeyer, Extending a flakiness score for system-level tests, in: IFIP International Conference on Testing Software and Systems, Springer, 2024, pp. 292–312.
[13] Meta Engineering Team, How do you test your tests? A Probabilistic Flakiness Score for Testing at Scale, 2020. URL: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/.
[14] S. Rasheed, J. Dietrich, A. Tahir, On the Effect of Instrumentation on Test Flakiness , in: 2023 IEEE/ACM International Conference on Automation of Software Test (AST), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 123–127. URL: https://doi.ieeecomputersociety.org/10.1109/AST58925.2023.00016. doi:10.1109/AST58925.2023.00016.
[15] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, Y. Le Traon, The importance of accounting for execution failures when predicting test flakiness, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1979–1989. URL: https://doi.org/10.1145/3691620.3695261. doi:10.1145/3691620.3695261.
[16] M. H. U. Rehman, Quantifying Flaky Tests to Detect Test Instabilities, Ph.D. thesis, Master's thesis. Concordia University. https://spectrum. library. concordia …, 2019.
[17] Anonymous, Fladage: A framework for generation of synthetic data to compare flakiness scores, 2025. URL: https://doi.org/10.5281/zenodo.17206909. doi:10.5281/zenodo.17206909.
[18] W. Lam, International Dataset of Flaky Tests (IDoFT), 2020. URL: http://mir.cs.illinois.edu/flakytests.
[19] P. Wendler, S. Winter, Regression-test history data for flaky-test research, in: Proceedings of the 1st International Workshop on Flaky Tests, FTW '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 3–4. URL: https://doi.org/10.1145/3643656.3643901. doi:10.1145/3643656.3643901.
[20] J. Wang, Y. Lei, M. Li, G. Ren, H. Xie, S. Jin, J. Li, J. Hu, Flakyrank: Predicting flaky tests using augmented learning to rank, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2024, pp. 872–883.
[21] A. Alshammari, C. Morris, M. Hilton, J. Bell, Flakeflagger: Predicting flakiness without rerunning tests, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1572–1584. doi:10.1109/ICSE43902.2021.00140.

A. Flakiness Assignment

A.1. Test-Level Flakiness Assignment

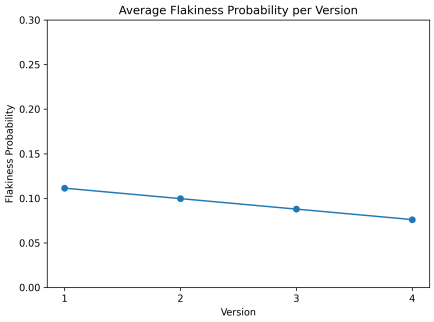
Figure 1: Average Test-Level Flakiness Probability per Test



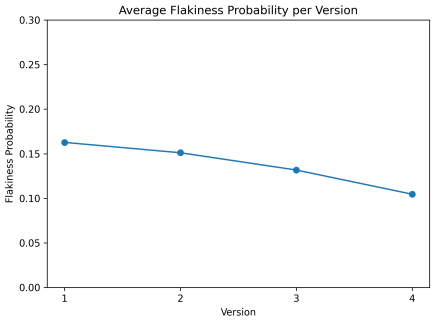
A.2. Version-Level Flakiness Assignment

Figure 2: Version-Level Flakiness Probability Averages

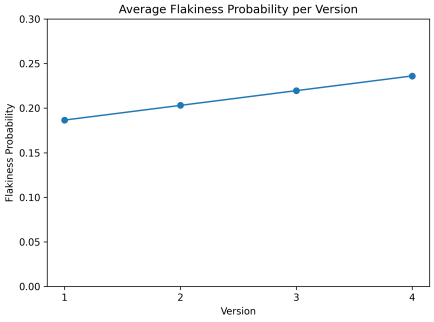
(a) Decrease



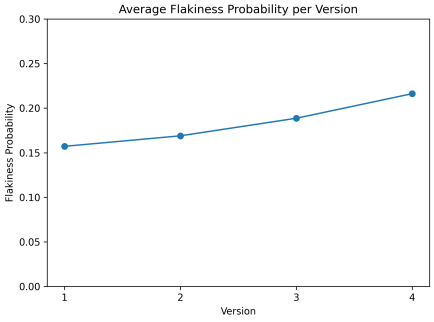
(b) Decrease exponential



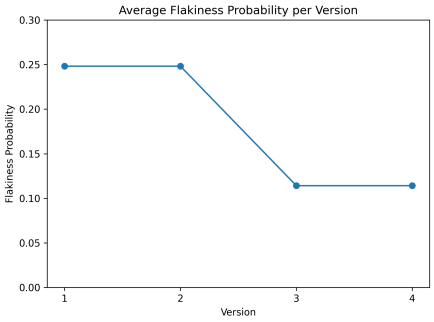
(c) Increase



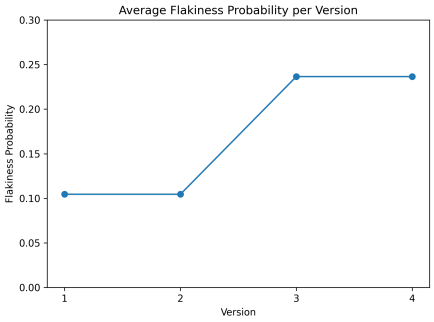
(d) Increase exponential



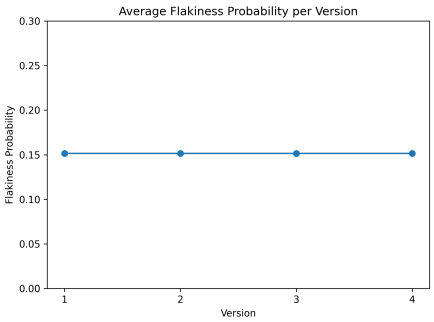
(e) Sudden decrease



(f) Sudden increase



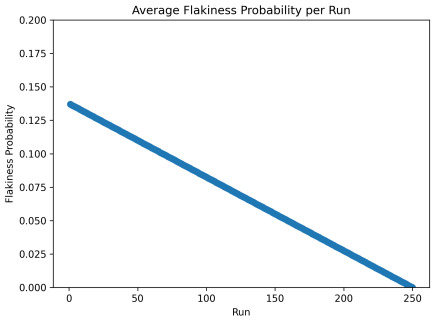
(g) Uniform



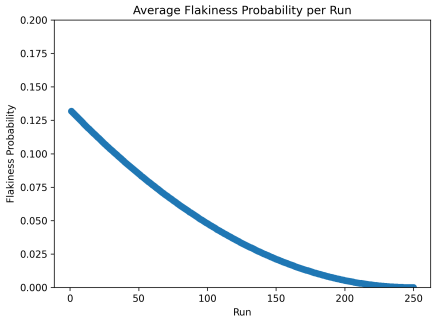
A.3. Run-Level Flakiness Assignment

Figure 3: Run-Level Flakiness Probability Averages

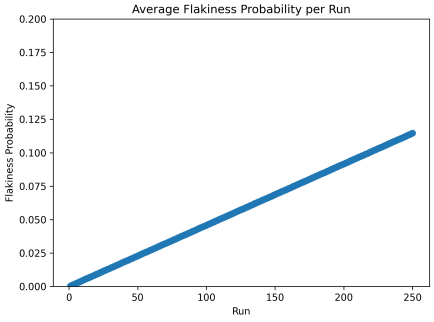
(a) Decrease



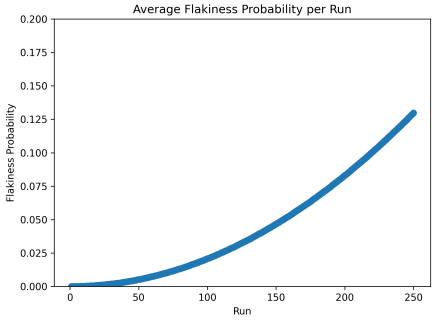
(b) Decrease exponential



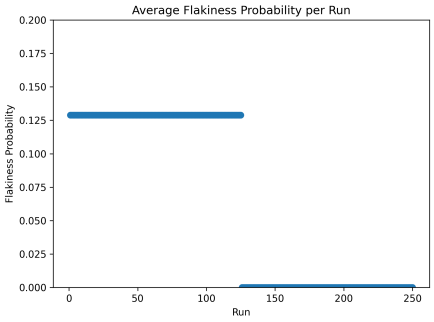
(c) Increase



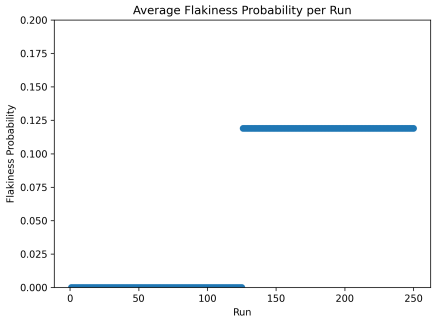
(d) Increase exponential



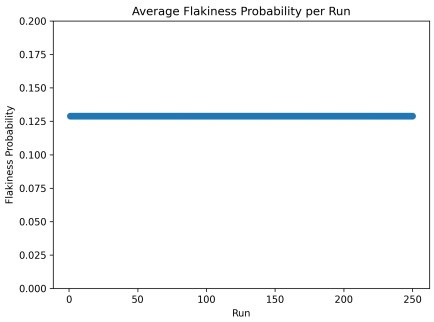
(e) Sudden decrease



(f) Sudden increase



(g) Uniform

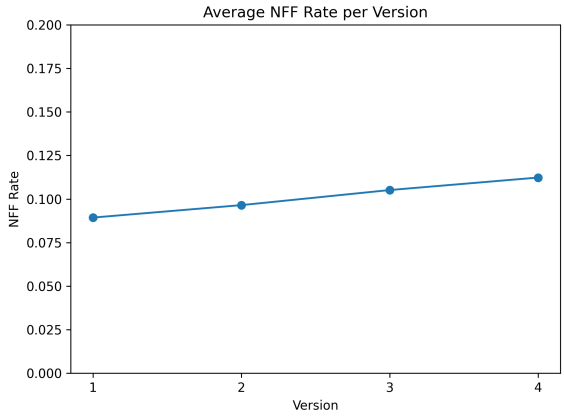


B. Algorithm Perspectives

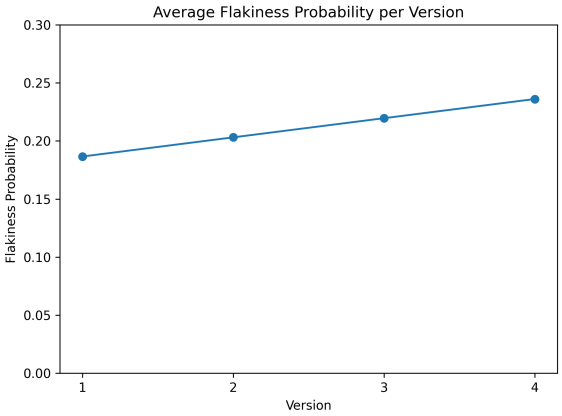
B.1. Trend Correlation with Ground Truth

Figure 4: NFF and Ground Truth Pattern Correlation

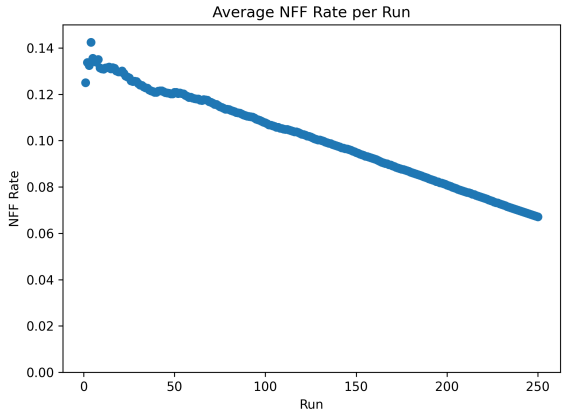
(a) Average NFF rate per version



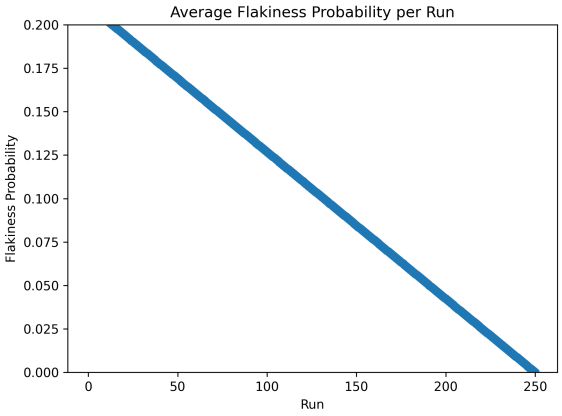
(b) Average flakiness probability per version



(c) Average NFF rate per run



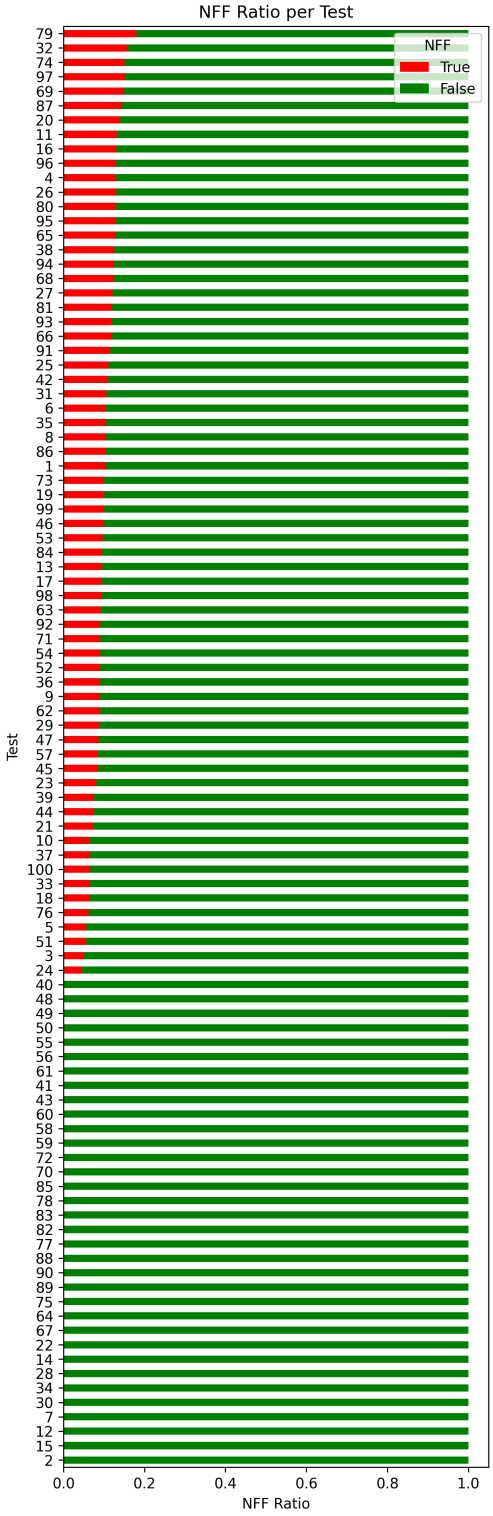
(d) Average flakiness probability per run



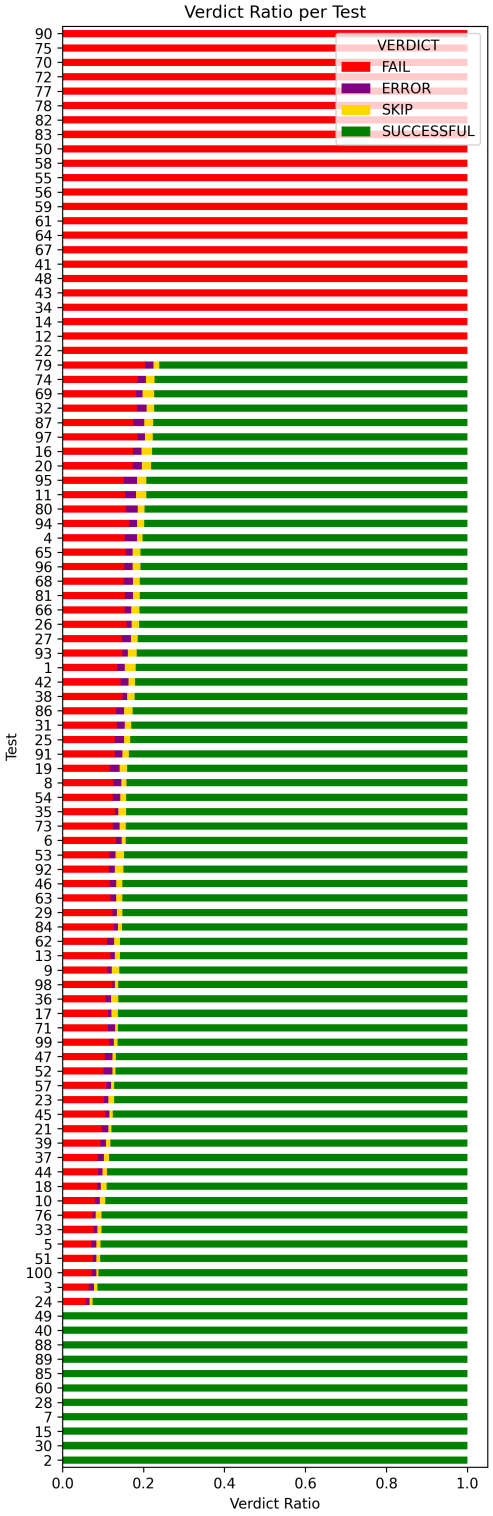
B.2. Outcome Correlation of NFF and EFS

Figure 5: Comparison of NFF and EFS Ratio per Test

(a) NFF ratio per test



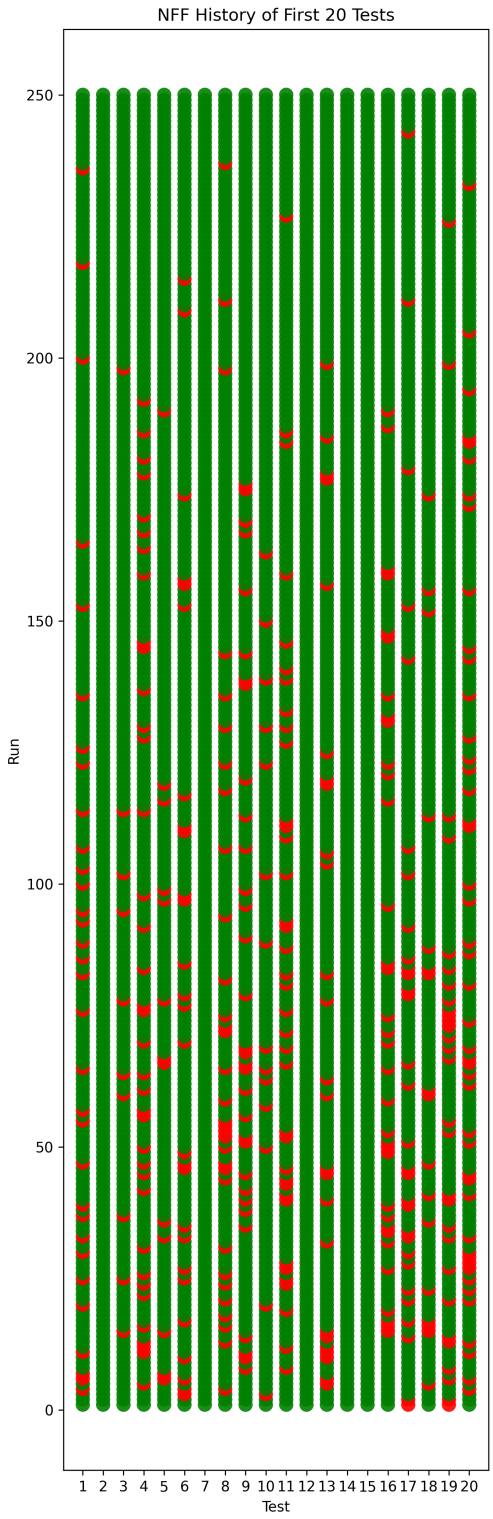
(b) EFS result ratio per test



B.3. History Correlation of NFF and EFS

Figure 6: Comparison of NFF and EFS Views of History for the First 20 Tests

(a) NFF View of History of First 20 Tests



(b) EFS View of History of First 20 Tests