Modeling and Ranking Flaky Tests at Apple

Emily Kowalczyk
Apple Inc.,
Cupertino, USA
ekowalczyk@apple.com

Karan Nair
Apple Inc.,
Cupertino, USA
karan_nair@apple

Zebao Gao
Apple Inc.,
Cupertino, USA
zebao_gao@apple.com

Leo Silberstein
Apple Inc.,
Cupertino, USA
lsilberstein@apple.com

Teng Long
Apple Inc.,
Cupertino, USA
teng_long@apple.com

Atif Memon
Apple Inc.,
Cupertino, USA
atif_memon@apple.com

ABSTRACT

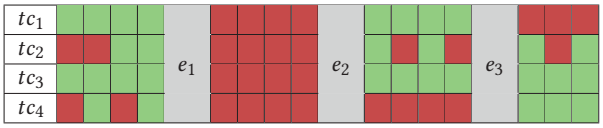
Test flakiness—inability to reliably repeat a test's Pass/Fail outcome—continues to be a significant problem in Industry, adversely impacting continuous integration and test pipelines. Completely eliminating flaky tests is not a realistic option as a significant fraction of system tests (typically non-hermetic) for services-based implementations exhibit some level of flakiness. In this paper, we view the flakiness of a test as a rankable value, which we quantify, track and assign a confidence. We develop two ways to model flakiness, capturing the randomness of test results via entropy, and the temporal variation via flipRate, and aggregating these over time. We have implemented our flakiness scoring service and discuss how its adoption has impacted test suites of two large services at Apple. We show how flakiness is distributed across the tests in these services, including typical score ranges and outliers. The flakiness scores are used to monitor and detect changes in flakiness trends. Evaluation results demonstrate near perfect accuracy in ranking, identification and alignment with human interpretation. The scores were used to identify 2 causes of flakiness in the dataset evaluated, which have been confirmed, and where fixes have been implemented or are underway. Our models reduced flakiness by 44% with less than 1% loss in fault detection.

ACM Reference Format:
Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381370

1 INTRODUCTION

"Continuous integration (CI) and testing" is the cornerstone of quality assurance in today's large companies [4, 11, 13, 14]. Developers integrate code into a shared repository several times a day. Each check-in is verified by an automated build-and-test process, fully integrated into the CI server, allowing developers to detect problems early [2, 10, 15, 17]. CI processes are severely hindered due to the presence of *flaky tests* [14, 15]. A flaky test is one that may fail or pass non-deterministically. Intuitively, a flaky test's outcome defies the developer's expectation. For example, if a developer runs a test case multiple times, keeping all things constant (the test code, source code, the environment, etc.), then the developer expects the test should constantly pass or fail as no apparent changes have occurred.

Table 1: Example test result history.



Consider 4 tests in Table 1. Each test is run 15 times, with passes (green) and fails (red) shown in order of execution from left to right. After 4 runs, we see *epoch*[1] e_1, which is a change to the code under test; e_2 is a modification to the test code that impacts all 4 tests; and e_3 is a change to the underlying data used by the software under test. Assuming that everything else remains constant between epochs, our expectation is that test results will not change between epochs (they may certainly change *across* epochs). This is not the case for tc_2 and tc_4. Both tests passed and failed non-deterministically before e_1; test tc_2 between e_2 and e_3, and after e_3.

Even though one can claim to control "everything" when re-running tests, flakiness exists for a number of valid—and prevalent—reasons. Network traffic and latency, asynchronous waits and concurrency are some of the common causes of test flakiness [3, 8, 12]. Google reported 16% of their 4.2 million tests were flaky, causing 1.5% of their test runs to flake [15]. Similarly, Microsoft analyzed 5 projects and found 4.6% of their tests were flaky [10], and Mozilla maintains a database of flaky tests, which they have estimated grows by more than 100 new flaky tests each week [3].

We recognize that flaky tests may need to exist in a test repository because they sometimes help to uncover real bugs – indeed the underlying causes of flakiness may point to bugs in the test infrastructure or the code under test. We however, focus on the CI usecase, where flakiness causes tests to provide undesirable signals that cause unnecessary delays. By and large, developers agree that

[1] A significant event such as a code/data/config/flags change that may cause a test to change its behavior.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381370

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

flaky failures in CI are not really actionable; in fact, if a developer re-runs a Failed flaky test and sees it Pass then the Failure will most likely be ignored.

Previous work has identified causes of flakiness [3, 12, 16] and automated certain flake handling tasks [1, 10, 14]. However, relatively little attention has been given to quantifying the flakiness level of a test. Quantifying flakiness is useful when evaluating large suites or where all tests have some degree of flakiness, a situation not uncommon in practice for non-hermetic tests (i.e., tests not run in pure isolation), such as system tests [15].

In this paper, we view the flakiness of a test as a measurable value, which we quantify, track and assign a confidence. We build upon prior work [6] by formally modeling and quantifying test flakiness using *entropy*, a measure of randomness, and *flipRate*, which models the temporal variation in test outcomes. We aggregate these metrics to provide a *flakiness score* per test case. We show how flakiness is distributed across tests for 2 large services at Apple, including typical score ranges and outliers. The flakiness scores are used to monitor and detect changes in flakiness trends. Our experiments show near perfect accuracy in ranking, identification and alignment with human interpretation. Our deployed service helped to reduce flakiness by 44% with less than 1% loss in fault detection, and the scores identified 2 causes of flakiness which have been confirmed, and fixes have been implemented or are underway.

In particular, we make the following new contributions:

- Formalized model to quantify the flakiness of a test.
- Demonstrate uses of scores for prioritization, filtering, and flakiness triage.
- First experiment evaluating accuracy of flakiness scores including a comparison with tester interpretation.
- Confidence measure of the scores computed.
- Measures of flakiness on two production environments.
- Impact of flakiness in a new context—integration tests for distributed services—with many controllable, uncontrollable, and partially controllable dependencies (data, other internal services within org, etc).

The next section discusses related work; Section 3 discusses our models of flakiness; we describe our datasets in Section 4 that we use in our experiments in Section 5. We finally conclude with a discussion of ongoing and future work in Section 6.

2 RELATED WORK

Researchers have identified various causes of flakiness. Luo et al. [12] identified 10 causes: async wait, concurrency, test order dependency, resource leak, network, time, IO, randomness, floating point operations, and unordered collections. Eck et al.[3] identified 4 new causes: restrictive value ranges, test case and suite timeouts, and platform dependency. Palomba and Zaidman [16] related test flakiness to test smells (anti-patterns in the test's code) for codebase JUnit tests.

Bell et al. presented *DeFlaker* to identify a flaky test run based on the test's coverage of changed code [1]. Lam et al. [10] developed *RootFinder* to assist triaging causes of flaky test runs by providing a ranked list of differences in the logs of passing and failing runs. Memon and Micco [14] suggested features such as the type of test (UI, integration, unit) and number of edges (i.e., flips) may be used to predict flaky tests. King et al. [9] suggested a model for predicting flaky tests using a Bayesian network, static test properties and past test results. Others have automatically identified single causes of flakiness such as concurrency [5] and test ordering [11, 18].

Relatively little attention has been given to quantifying the flakiness of a test, which is the focus of our work.

Gao et al. used an entropy metric to study the impact of external factors on the repeatability of system UI tests [7]. They measured the stability of 3 artifacts – code coverage, invariants, and GUI state– and later used the metric to rank flaky tests and false UI test alarms on single SUT versions [6].

We build upon the work of others in that we look at patterns of test results for our models. Unlike previous work, our model summarizes a test's behavior over time and multiple versions, and we perform the first extensive study on the model's accuracy. We provide the first detailed comparison of the differences between entropy and flip rate for measuring flakiness, and make the first steps toward formalizing flakiness to apply to modern complex services. We also define a confidence value to our scores to provide an additional signal about the score's reliability and assist automated interpretation.

3 MODELING FLAKINESS

We now provide more details on how we model and quantify flakiness, using the test cases from Table 1 as a running example. Intuitively, our model works by grouping the result (passes and fails) history of a test into *versions*; we expect a test's result to be the same (i.e., all passes or all fails) inside a version. If a test result varies inside a version, we call the test flaky. We then quantify the variation of results inside a version using two scores: flip rate and entropy. We then aggregate these scores across versions to provide a value quantifying the overall flakiness level of the test.

3.1 Versions

A central concept to our model is a *version* that determines the sequence or set of results within which to measure variability. Each version is composed of *version factors* – components of the software under test (SUT), flags, configs, test code, and infrastructure. If these factors remain unchanged, we expect a test's results to be repeatable, and all exceptions are considered flakes. In our running example in Table 1, our epochs were changes to code, test code, and data. For convenience, lets assume that our version is also composed of these 3 factors: the code, test code, and data. Each time a factor changes a new version is created. This results in 4 versions in our example, version v_1 are all runs before e_1; v_2 is between e_1 and e_2; v_3 is between e_2 and e_3; and v_4 is after e_3.

We see 2 tests have varying results within versions, and are therefore flaky. tc_2 varies on v_1, v_3; and v_4; and tc_4 varies on v_1. All other tests are non-flaky. If we change the version definition for the example, the tests identified as flaky would change. For example, if only code and test code were included, then tc_1 would also be considered flaky, since the failures caused by data in v_4 would be processed as flakiness.

The use of customizable versions allows teams to factor out controllable components of the SUT, test or infrastructure from the model's scores. For example, for unit tests, which often live in the

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

same repository as the code they test, the version may consist of only one factor, the code commit id of the SUT. Integration and system tests may contain more factors, for example, if tests live outside the code repository or the SUT contains multiple components spread across repositories, then multiple commit ids may be needed. If the SUT has a dependency on an internal data build, the data build id can be included as a version factor.

3.2 Quantifying Flakiness Inside a Version

We now formalize how flakiness inside a version is quantified. Suppose we have a CI environment where new versions of a SUT are deployed and tests are executed. Each time a test is executed it gives a result, which accumulate over time into a result history. We refer to such a history as $R = (r_{v_1,t_1}, ..., r_{v_1,t_m}, r_{v_2,t_{m+1}}, ..., r_{v_n,t_n})$, which contains the results of a test on a specific environment indexed by both the version it ran on v and the time it was executed t.

In practice, a test may run multiple times on the same v. For example, if a strategy is implemented to identify flaky failures or if tests are run at regular intervals (for example, hourly). We refer to these as *re-runs*, or $R_{v,*} = (r_{v,t_1}, ..., r_{v,t_n})$, where $R_{v,*} \subset R$ and contains all results of a test on a specific v.

At the core of our model is a function $f : (R_{v,*}) \rightarrow [0, 1]$, which quantifies the flakiness of results in $R_{v,*}$. f outputs a value between [0,1] indicating the flakiness level of the results on v, where 0 indicates non-flaky and 1 most flaky.

Our first measure of flakiness is based on *entropy*, a metric from information theory to quantify the disorder or uncertainty of a random variable. With respect to this work, we use entropy to measure the uncertainty of a test's result on v.

$$entropy(R_{v,*}) = -\sum_{i \in \{P,F\}} p(i) \log p(i), \tag{1}$$

where $p(i)$ is the probability of result type i based on the results of $R_{v,*}$. When $p(i) = 0$, we treat $log_2(0)$ as 0. Because we use a base of 2 and have only 2 unique results (pass, fail), the metric outputs a value between [0, 1]. A test which always passes or fails on v has a score of 0, and a test which fails and passes on half the runs has a score of 1.

In our running example, we can compute the entropy on v_1 for each test. tc_1 and tc_3 have $p(P) = 1$ and $p(F) = 0$, and therefore have entropies of 1. tc_2 and tc_4 have $p(P) = 0.5$ and $p(F) = 0.5$, resulting in entropies of $-2(0.5 * log_2(0.5)) = 1$.

We see tests tc_2 and tc_4 have identical scores for version v_1, even though tc_4 flips at every run and tc_2 remains more constant in v_1. To capture these differences, we need a model that quantifies the variation in results in the order they were run. We now define our second metric called *flipRate* that measures the rate at which a test's result flips between Pass and Fail in $R_{v,*}$:

$$flipRate(R_{v,*}) = \frac{numFlips(R_{v,*})}{numPossibleFlips(R_{v,*})}, \tag{2}$$

numFlips is the number of transitions between P→F and F→P in the $R_{v,*}$, and *numPossibleFlips* is the total number of possible transition points (i.e., pairs of consecutive runs).

If we return to our running example, we can now compute the flip rate for each test on v_1 We can see tc_1 and tc_3 never flip, so their flip rates are 0. tc_2 flips once from P→F between r_2 to r_3, and therefore, has a flip rate of $1/3 = .33$. tc_4 flips 3 times from r_1 to r_2, r_2 to r_3, and r_3 to r_4, and therefore, has a flip rate of $3/3 = 1$. We can see tc_2 and tc_4 have different flip rates, which differ by .67. So, flip rate was able to capture the differences in the tests' behavior, which entropy could not.

3.3 Aggregating Flakiness Across Versions

Function f measures the flakiness of a test inside a *version*. We now aggregate these to quantify flakiness of the *test*. We explore two modes of aggregation. The first is an *unweighted* model U, which treats all versions in the time window equally:

$$U_H(R) = \sum_{v \in V^t_{t-H}} \frac{f(R_{v,*})}{|V^t_{t-H}|} \tag{3}$$

Given a result history R and history length H, the model computes the average flakiness measured by f for all versions in the last H days, V^t_{t-H}.

The second is a *weighted* model W that computes an *exponentially weighted moving average*, giving more weight to outputs of f on more recent versions:

$$W_{\lambda,P}(R, n) = Z_n = \frac{\lambda x_n + (1 - \lambda) Z_{n-1}}{\lambda \sum_{i=0}^{n-1} (1 - \lambda)^i} \tag{4}$$

Given a result history R and two parameters, the decay value λ and period P, the model computes a score, Z_n, for a discrete time step n where $n = \lceil t/P \rceil$. Z_n is based off of the score of previous periods, Z_{n-1}, plus the flakiness of re-runs on versions in the period of the current time step, $x_n = \sum_{v \in V^{nP}_{(n-1)P}} f(R_{v,*})$. In order to determine the version in the current period, a mapping between v and time must exist. This can be implemented in several ways such as the version's time of the last run or time of deployment. We use the latter for ease of implementation, where the latest version is always assigned max weight and weights decay based on distance from that version. The model is then normalized using the sum of weights across time steps $\lambda \sum_{i=0}^{n-1} (1 - \lambda)^i$.

If we return to our running example, we can now compute aggregated scores for each test using each model. Since tc_1 and tc_3 have never flaked, they receive a score 0 from all models, indicating they are non-flaky. If we assume all versions run in a consecutive days, H is 3 days, and λ is 0.1, tc_2 would have the following scores: $(0.33 + 0 + 1)/3 = 0.44$ on U_{FR}, $(1 + 0 + 1)/3 = 0.67$ on U_E, $(1 + 0.9^1 * 1 + 0.9^2 * 0 + 0.9^3 * .33)/(1 + 0.9^1 + 0.9^2 + 0.9^3) = 0.62$ on W_{FR} and 0.76 on W_E. tc_4's scores would be computed similarly, resulting in 0.33 for U_{FR} and U_E, and 0.21 for W_{FR} and W_E. Entropy and flip rate models give tc_4 the same score, because the only version the test was flaky on (v_1) had the same flip rate and entropy score of 1.

3.4 Confidence

For each flake score, we also compute a confidence value. We base our confidence on two factors: number of re-runs used to compute the score, and the recent variability in the score's value. This allows us to identify when the model lacks sufficient data, or a test's behavior is changing and the score unrepresentative.

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

We compute several measures of confidence. First, we compute the standard deviation σ of each test's score using a sample over the last d days. This provides a measure of the recent variation in each test's scores. σ is used to compute an additional measure, margin of error (MOE). We use MOE as merely a useful measure to further scale σ based on the amount of data collected for the test. The MOE is computed as $t * \sigma/\sqrt{n-1}$, where t is the t-statistic for a defined confidence level, σ is the standard deviation of the sample's scores, and n is the size of the sample. In general, the smaller the σ and MOE, the more confident the model is in the score.

If scores are not computed regularly, the number of re-runs can be used as a confidence instead. This does not capture the score's variability, but does help identify scores where the model has too little data. The number of re-runs can be normalized to provide a value between [0, 1] as $1 - 1/\sqrt{n}$, where n is the number of re-runs. Minimum thresholds can then be set to ignore scores with insufficient data.

We use confidence values in several ways. First, a minimum threshold for the number of re-runs is set to assure scores have a reasonable amount of data. Thresholds are typically set to around 10 re-runs, but can be tuned based on the number of re-runs in the dataset. Once tests pass this threshold, standard deviations or MOEs are used to compute lower bounds of scores by subtracting the MOE from the score. This provides a conservative estimate of the score. Prioritization or filtering can take place on the lower bounded score, which incorporates the confidence.

We now compute the confidence values for our flaky tests, tc_2 and tc_4, from the running example. First, we compute the number of re-runs. Since both tests run 15 times on 4 versions, the tests have 11 re-runs in total. If we assume scores were computed after each run of the test, this would result in a score history of 15 scores. We can compute σ and MOE using the history. For the U_{FR} models, σ for tc_2 would be 0.128 and 0.187 for tc_4. σ can then be used to compute the MOE, which would equal $1.76 * \sigma/\sqrt{(15-1)}$ where 1.76 is the t-statistic for a 95% confidence level and degrees of freedom equals the size of the score sample minus 1. This results in MOE= 0.06 for tc_2 and MOE=0.09 for tc_4. Since tc_2 and tc_4 have the same amount of data, the difference in their confidences is due to variability. tc_2 has higher confidence, because it flakes more consistently than tc_4.

4 DATASETS

Before we present our experiments (Section 5), we describe the datasets of test results (Pass/Fail) used therein. The datasets include results from test suites for 2 services at Apple, S_1 and S_2, and artificial labeled data that we created.

4.1 Datasets R_1, R_2, R_3 from Production Services

Our production datasets contain results for suites of functional integration tests, written as parameterized *test frames* – each is a template of a test whose parameters are instantiated into numerous concrete runnable test cases. All tests run in their respective service's CI pipeline, broken into environments representing various stages in the release cycle, such as pre-merge, post-merge, and staging. Each environment may differ in ways that impact tests. Some possible differences include: churn rate (i.e., release cycle of versions), differently configured machines, infrastructure choices (VM vs. hardware), location of data centers, and load.

The test results for S_1 are from a single CI environment E_s. Tests run hourly on the latest deployed version, updated every week. Tests for S_2 run in 3 different CI environments, E_{pm}, E_d, E_w. We create three datasets (R_1, R_2, and R_3) from S_1 and S_2. The R_1 dataset spans 1 month of test results for the S_1 service. It contains 4.9M results for 6.4K tests and 88 frames, where results are from a single suite and environment, E_s. The R_2 dataset contains results for the S_2 service. It is the largest dataset with results spanning 3 months, and contains roughly 99M results for 99K tests and 540 test frames. Unlike R_1, the dataset contains multiple suites and environments, which allows us to inspect various differences between each.

Our R_3 dataset is coarsely labeled from S_2. A few months before the writing of this paper, S_2 developers reported an increase in flakiness in 1 of their 3 suites in a particular environment. We collected results for all suites on the environment for the reported period as well as the preceding month. The resulting dataset spans 5 weeks (4 weeks before the increase and 1 week after), and contains 1.1M results for 45K tests.

4.2 Artificially Labeled Dataset (A_1)

Our second dataset allows us to evaluate the accuracy of our models by having a ground truth to compare the output against. Since the model quantifies how flaky a test is, the ground truth indicates not only whether the test *is* in fact flaky, but also *how* flaky.

We implemented a generator to create a dataset of artificial results with known flakiness labels. The generator models flakes as probability mass functions (PMF), where a test has a probability of flaking on any given run, and a given version of the SUT has a probability of containing a fault detectable by the test, which causes it to fail deterministically.

Given a run schedule, which defines the churn of versions on the simulated environment and the number of test runs, the generator simulates running the schedule using the PMFs to determine the results within each version. Once finished, the created history should encode the flakiness level of the test. It contain no flakiness for tests where the probability of flaking is 0, and max flakiness when 1. Max flakiness is defined as a test transitioning between pass and fail on every run.

Algorithm 1 describes the algorithm in more detail. The generator creates a result history for a single test. It accepts a *testId* to uniquely identify a test, *runSchedule* containing version ids and the number of times the test runs on that version, P_{fault} the probability the new version contains a fault detectable by the test, P_{flake} the probability the test will flake on any given run. The generator uses P_{flake} and P_{fault} to create binomial distributions D_{flake} and D_{fault} which, when sampled, will flake or fault with probabilities P_{flake} and P_{fault} respectively. For each version in the *runSchedule*, the generator samples D_{fault} to determine whether the new version contains a fault detectable by the test. If a fault exists, all runs on that version are marked as fails. This is done because we assume a test cannot (or is highly unlikely to) flake into a passing state. Alternatively, if the version is correct, the generator samples D_{flake} for each run on the version. If the test flakes the

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

generator flips the result, otherwise it remains the same as the previous run. This results in a test with $P_{flake} = 0$ never changing the result from the faultState, and a test of $P_{flake} = 1$ flipping every run. The generator then creates a record for each run saving the *testId*, *versionId*, *runId*, *startTime*, and *result* to the history, and repeats until the *runSchedule* is complete.

Algorithm 1: Generate tests results with flakiness.

```
Data: test id, run schedule, P_flake, P_fault.
Result: Artificial results with seeded flakiness.
D_flake ⟵ Binomial(P_flake);
D_fault ⟵ Binomial(P_fault);
resultHistory ⟵ ∅;
foreach version in runSchedule do
    isFaulty ⟵ sample D_fault;
    result ⟵ resolve isFaulty to pass or fail;
    foreach run on version do
        if not isFaulty then
            didFlake ⟵ sample D_flake;
            if didFlake then
                result ⟵ flipResult(result);
            end
        end
        record ⟵ Create record with test, version, run and
          result data;
        resultHistory.append(record);
    end
end
return resultHistory;
```

We used the generator to create test histories for 120 tests, where 80% are flaky and 20% are non-flaky. The *runSchedule* remained constant across tests, and was based off the average runs and churn in the E_d CI environment in S_2. Each test simulated running a total of 1,500 times with 267 versions where each version had between 1-20 runs. We modeled the fault PMF from the same historical data. We found 55% of tests never fail and 45% fail between 0.05-0.4% of runs. We constructed the artificial dataset similarly, where 55% of tests were assigned a P_{fault} value of 0, and 45% a random value between [0.005, 0.04]. For the non-flaky tests, P_{flake} values equaled 0, and for flaky tests, they uniformly covered (0, 1], where each test differed by a small constant. Since we generated 100 flaky tests for our experiment, this resulted in values spanning [0.01, 1] and differing by 0.01.

All generated results histories were combined into a single dataset, which we refer to as A_1. Each record contained the *testId*, *versionId*, *runId*, simulated *startTime* for the run, and a *result* of pass or fail. We recorded the parameter values (*testId*, P_{flake}, P_{fault}) used to generate the results in a separate table. These define the labels (P_{flake}) and keys (*testId*), which are used to evaluate the model's output against. P_{fault} is saved to help manual analysis.

Each of the datasets are used to answer the RQs described in Section 5 and listed in Table 2. A_1, the artificial data is used to evaluate parameters and model accuracy in RQ1. R_1 and R_2 are then used in RQ2-4 to quantify and evaluate the models usefulness. Finally, R_3 is used in RQ5 to evaluate flakiness trends.

5 EXPERIMENTS

We now describe the experiments and metrics used to evaluate our models of flakiness. Because our models are parameterized, we have designed our experiments to first identify the most effective parameter values for each model type; we then apply the best performing models to production data to gather insights about the flakiness of tests and CI environments under study; finally we evaluate the usefulness of our models in various flake handling tasks such as filtering flaky tests, identifying flakiness causes, and analyzing flakiness trends.

We look to answer the following research questions:

RQ1: (*Accuracy & Model Selection*)

RQ1.1: How accurate are the models at identifying and ranking flaky tests? Which model types perform best (we used various measures of accuracy) and at what values for H and λ? What are the differences between weighted vs unweighted models and flip rate vs entropy models?

RQ1.2: How do model scores relate to human's understanding of flakiness? Which model types perform best?

RQ2: (*Quantifying Impact*) How much does test flakiness impact the CI pipelines in our study? How is flakiness spread across tests in each?

RQ3: (*Identifying Causes*) Can models identify causes of test flakiness in production data?

RQ4: (*Reducing Flakiness*) Can models reduce test flakiness? How much can be reduced and at what cost to fault detection?

RQ5: (*Detecting Changes in Flakiness Trends*) Can time series of flake scores be used to detect changes in flakiness trends?

5.1 RQ1.1: Artificial Test Results

To test and optimize each model's accuracy, we first ran experiments on the artificial data described in Section 4. Artificial data allowed us to compare the model's output against a ground truth, and evaluate the impact of parameters such as history length H and decays λ as well as model type.

Procedure & Metrics: We ran a grid search for each model type. Table 3 summarizes the parameters and value ranges searched for each. Agg. indicates whether the model is weighted W or unweighted U; f is the function used to quantify flakiness, where E is for entropy and FR flip rate; H indicates the value ranges searched for H in unweighted models; λ is the range of λ values searched for weighted models; Step is the step function used to iterate the parameter ranges; and i is the input to the step function. Unweighted models searched for values of H between 1-64 days with steps of 2^i days for i between $[0, 6]$. Weighted models searched for values of λ between $[0.1, 0.6]$ with constant steps of 0.1. P remained constant at 1 day.

For each model *configuration* (choice of model type and parameter values) the accuracy was measured for two tasks: identification and ranking. Identification measures each model's ability to distinguish flaky tests from non-flaky tests, while ranking measures its ability to quantify and rank flaky tests. We first computed the

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

Table 2: Datasets used in our study. Type lists the type of tests in the suite, where Func. is for functional. SUT is the service under test. Labels indicate if all tests in the dataset contain flakiness level labels. A star means data is partially labeled. For example, R_2 has the top 15 tests labelled, and R_3 has flakiness at the aggregated suite level labeled. Envs. lists the number of environments results are collected for. # Tests and # Frames are the number of unique tests and frames which ran respectively. Dur. is the length of time the results cover. RQs are the research questions each was used to answer.

Name	Source	Type	# Suites	SUT	Labels	Envs.	# Tests	# Frames	# Results	Dur.	RQs
A_1	Artificial	-	-	-	✓	-	125	126K	-	6 wks	RQ1
R_1	Real	Func.	1	S_1	✓*	E_s	6.4K	88	4.9M	1 mos	RQ1-2
R_2	Real	Func.	3	S_2	-	E_{pm}, E_d, E_w	99K	540	99.7M	3 mos	RQ2-4
R_3	Real	Func.	3	S_2	✓*	E_d	45K	680	1.1M	6 wks	RQ5

Table 3: Model types and searched parameter values.

			Searched			
Model	Agg.	f	H (days)	λ	Step	i
W_{FR}	W	FR	-	[0.1,0.6]	i	0.1
W_E	W	E	-	[0.1,0.6]	i	0.1
U_{FR}	U	FR	[1,64]	-	2^i	[0-6]
U_E	U	E	[1,64]	-	2^i	[0-6]

accuracy of the model for identifying flaky tests. A test was labeled flaky if its P_{flake} value was above 0, and non-flaky otherwise. We marked true positives (TP) as tests labeled as flaky where the score was greater than 0; true negatives (TN) were tests labeled as non-flaky with scores equal to 0. Accuracy *acc* was then computed where the denominator was equal to the total number of tests.

Next we measured the model's accuracy in ranking by computing Spearman Rank Correlations between the model's scores and P_{flake}, the probability used to generate the result. A Spearman Rank Correlation is a measure used to assess whether two variables monotonically increase together. Its correlation coefficient, ρ, provides a value between [-1, 1], where 1 indicates perfect correlation, 0 uncorrelated, and -1 inversely correlated. For this work, a ρ value of 1 indicates the most accurate model, where the model's rankings are identical to the true rankings of the tests.

The top configurations for each model type were identified, where top models were those with *acc* and ρ values closest to 1. To determine if the correlation was significant, we ran a two-sided hypothesis test for each top configuration. Since multiple comparisons were made, Bonferroni corrections were applied, and $p < 0.025/n$ indicated significance where n was the number of hypothesis tests. We then manually evaluated the results, and investigated how parameter values impacted the model's *acc* and ρ within and between model types.

Results:

Figure 1 shows the impact of parameters H and λ on the accuracy of the models. The x-axis is the range of parameter values searched, where parameters are ordered so model configurations from left to right have increasing memory (i.e., models using the parameter value compute scores using longer histories or giving higher weight to past results due to small decay). Solid lines are values of ρ, the measure of the model's ranking accuracy, while dotted lines are *acc*, the accuracy of the model at identify flaky tests. Color differentiates flip rate and entropy models, where orange lines indicate results for flip rate models, and blue lines entropy. The top plot displays results for unweighted models and the bottom weighted.

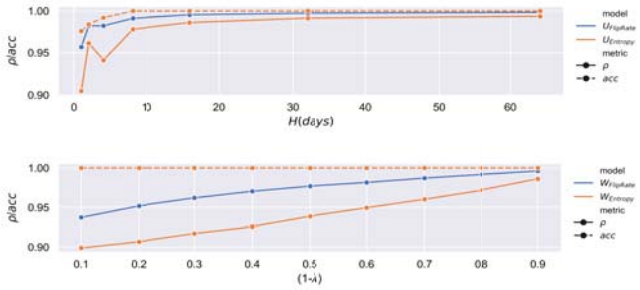


Figure 1: The impact of parameters on each model's accuracy.

We note that overall, most configurations of the models performed well with *acc* ranging from [0.98, 1], and most are able to detect all flaky tests. No difference in *acc* existed between entropy and flip rate models, however, weighted and unweighted models differed. Unweighted models with short histories (1-8 days) misclassified flaky tests as non-flaky. The misclassified tests had P_{flake} values between [0.01, 0.03], and did not flake in the time window scores were computed over. Weighted models did not suffer from this, because even though they devalue old flakes, some small value persists and keeps it above 0.

All models showed strong correlation with probabilities used to generate the results. ρ values ranged from [0.9,1]. Flip rate models slightly outperformed entropy models with ρ values 0.01 or higher. The best performing models were U_{FR} when H was equal or greater than 32 days, and W_{FR} with λ near 0.1. Both configurations achieved near perfect ρ values of 0.997 and 0.996, and their p-values indicated significance (1.51e−12 and 2.045e−131). Window sizes between 14-32 days were not optimal, but still performed well with ρ values near 0.99. Weighted models showed larger differences between values searched, but flip rate models with λ between [0.2,0.3] also performed well with ρ values near 0.99. Unweighted models saw a dip in ρ when H was equal to 4. A period of flakiness, not included when H equaled 2 and de-emphasized with additional data when H equaled 8, caused the model to overestimate flakiness at the time giving less correlated results.

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

Figure 2 shows a more detailed view of the results. Each sub-plot compares the model's scores with the true flakiness level of the test s_{true}, where s_{true} equals P_{flake}. Each point in the plot is a score for a test based on the given model. A diagonal line indicates an identical mapping between the model and true scores. Color indicates the parameter value, where dark blue indicates scores from models with the longest memory and red the least.

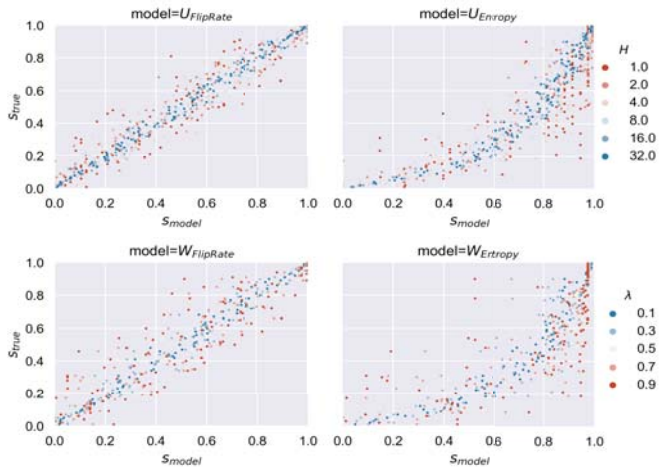


Figure 2: Model vs. true scores across searched parameter values.

We can more clearly see how larger memory improves the model's scores, which is not captured in the small differences in ρ and *acc*. Points in blue from models with longer memories are more ordered and monotonically increasing then points in red from models with shorter memory. Some unexpected patterns can be seen in low memory entropy models. U_E models with $H = 1$ push tests with s_{true} between [0.6,1] to 1 as indicated by the red points forming a vertical line in the top right corner of the plot. Many tests are overestimated with tests between [0.2,0.4] given scores between [0.8,1].

Differences between entropy and flip rate can also be observed. Models using entropy are slightly curved suggesting a non-linear relationship with s_{true}. As a result, some differences in mapping occur. Tests with scores between [0.2,0.4] are mapped to roughly [0.5,0.6]. Flip rate shows a more linear relationship with s_{true}, which is indicated by its diagonal line across the center of the plot. This may be because it is more closely related with the definition of flakiness.

Overall, all models showed high accuracy in identifying and ranking flaky tests on artificial data. Flip rate models with longer memory performed best, with the best configurations for U_{FR} with H near 32 days and W_{FR} with λ equal to 0.1. Both achieved perfect accuracy in identification (*acc* equal to 1) and near perfect in ranking (ρ and p-values equalling (0.997, 1.51e−12) and (0.996, 2.045e−131)). These results help us answer **RQ1.1**.

Table 4: A comparison of model scores to human labels.

		Scores<>Labels			
Model	d	NF	SF	F	VF
U_{FR}	0	-	[.09, .1]	[.11, .31]	[.5, .5]
U_E	0	-	[.13, .17]	[.18, .48]	[.64, .64]
W_{FR}	0	-	[.09, .09]	[.09, .21]	[.3, .3]
W_E	0	-	[.16, .17]	[.17, .45]	[.48, .48]

5.2 RQ1.2: Real Test Results

In addition to testing the models on artificial data, we also evaluated their accuracy on real labeled test results. This allowed us to confirm that models could perform regardless of intricacies not captured by the artificial data as well as assess how scores mapped to human's understanding of flakiness.

Procedure & Metrics: Models were run on the R_1 dataset, and the top 15 tests for each model were sent to the S_1 testing team for evaluation and labeling. Each model's parameters were tuned based on the results from Section5.1 with $H = 21$ days, $\lambda = 0.1$, and $P = 1$ day. These parameters are used in all remaining experiments (RQ2-5). Testers assigned a rank of *not flaky* (NF), *slightly flaky* (SF), *flaky* (F) or *very flaky* (VF) to each test based off their independent analysis of result histories and fault reports.

We compared the ranked labels by models and testers, and then inspected the mapping between scores to labels. Rankings of labels based off model and tester scores were then constructed, and the hamming distance between each was computed. To better understand how scores relate to human's interpretation of flakiness, we also looked at the mapping between labels and scores, and compared how ranges differed for each model type. Such information helps to interpret the model's output, and determine which values indicate a flaky, not flaky or very flaky test.

Results: Table 4 presents the results. The first column, Model, indicates the model evaluated, and the second column, d, shows the hamming distance between the tester's and model's labels. The remaining columns under Scores<>Labels provide more detail on the mapping between scores and tester labels. The table lists the range of scores, which corresponded to the tester's label of NF, SF, F, and VF.

We see that all models performed well with hamming distances of 0. This indicates there were no disagreements between the model and testers, and the model's output perfectly aligned with testers' interpretation. With respect to the mapping of labels to scores, as expected scores monotonically increase with the increasing flakiness label. Tests move from slightly flaky (SF) to flaky (F) around 0.1 for flip rate models and 0.17 for entropy. Tests move from flaky (F) to very flaky (VF) at roughly 0.5 for U_{FR} and W_E, while U_E began higher at 0.64 and W_{FR} lower at 0.3. Some ambiguity exists on the borders of classes. For example, W_{FR} and W_E models give scores of 0.09 and 0.17 for both SF and F tests. However, this is somewhat expected near label cutoffs.

Further differences in score values between model types can be seen. Entropy models have higher scores than flip rate, with consistently higher ranges across all classes. Differences in the distribution of ranges among labels also existed. W_{FR} maps VF tests between [0.3, 1.0] leaving [0, 0.3] for the other 3 classes. Other

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

models mapped VF to around or above 0.5 leaving a larger range for the other labels. Overall, however, these differences are relatively minor, and we report them to help others better interpret the model's scores.

All tuned models provided rankings that aligned with tester interpretations of flakiness. Minor differences in the mapping of scores to labels for each model were observed, however, the importance of these differences is a topic for future work and may vary by use case. These results help us answer **RQ1.2**.

5.3 RQ2: Quantifying the Impact of Flakiness

Next, we applied the models to the R_1 and R_2 datasets described in Section 4 to quantify the impact of flakiness on the suites and environments studied. Impact was assessed by computing the percentage of tests and failures that were flaky as well as evaluating the distribution of scores for flaky tests. Together these provided an estimate of the amount of flakiness at the test result level, as well as an indication of how each test was contributing.

Procedure & Metrics: To compute the percentage of flaky tests, we used the W_{FR} model. We took the latest scores for each test at the end of R_1 and R_2, and computed the percentage that were flaky. Percentage of flaky tests were computed as $numFlakyTests/numTests$, where a test with a score greater than 0 was flaky; $numTests$ was the sum of flaky and non-flaky tests, where a test was considered non-flaky if its score was 0 after more than 10 total re-runs. The run condition was placed on non-flaky tests to avoid including tests with too few runs to be classified.

We also computed the percentage of flaky failures as $numFlakyFailures/numFailures$, where $numFlakyFailures$ was the number of flaky failures, and $numFailures$ was the total number of failures seen across all runs in the dataset.

Finally, we evaluated the distribution of flakes scores for flaky tests to better understand how flakiness was spread across tests. To do this, we computed the distribution's quantiles (25%, 50%, 75%) to determine where the majority of scores for flaky tests fell. We also ran basic outlier detection using the interquartile rule (IQR).

Results: Table 5 presents the results. %FT is the percentage of flaky tests in each dataset, while %FTF is the percentage of flaky test frames. %FF is the percentage of failures that are flaky. The results show that overall, flakiness impacted many tests and frames in the suites. The model found 58% of tests were flaky in R_1 and 39% in R_2. Flaky tests were not constrained to specific frames with nearly all frames in R_1 having at least 1 flaky test, and 71% of frames flaky in R_2. These tests lead to roughly 12 and 16% of the failures in each CI pipeline being flakes.

Table 5: Quantifying test flakiness in R_1 and R_2

DS	$Q1$	$Q2$	$Q3$	Outliers	%FT	%FTF	%FF
R_1	0.002	0.003	0.004	443	58%	93%	16%
R_2	0.05	0.11	0.17	3,318	39%	71%	12%

Evaluating the distribution of scores for R_1 and R_2, we see tests in R_2 have higher scores than R_1. The majority of flake scores ranged from [0.002, 0.004] for R_1, while scores were higher in R_2 ranging from [0.05, 0.17]. This indicates that while R_1 had a high percentage of flaky tests, many of its tests were flaking infrequently. Such information may be useful for identifying causes, which we explore further in RQ3.

Scores were also used to identify outliers, where outliers were tests with abnormally high levels of flakiness within each dataset. The outliers column in Table 5 indicates the number of outliers detected R_1 and R_2. We can see in several hundred existed in R_1 and thousands in R_2.

In summary, we find over 50% of the tests in R_1 and 39% of tests in R_2 are flaky. With suites containing thousands of tests, these percentages motivate the need for a method of identifying worse culprits. By evaluating the distribution of flake scores, we see how flakiness was spread across tests for each pipeline, which could be used to identify common scores ranges and outliers. Such information may be useful for reducing and triaging flakiness, which we explore in **RQ3**. These results help us answer **RQ2**.

5.4 RQ3: Identifying Causes

We evaluated the impact of 2 factors—environment and test suite—on the flakiness in R_2. We evaluated R_2 only, because R_1 contains only 1 environment and suite.

Procedure & Metrics: To assess the impact of environments and suites, we compared the distribution of flaky test scores for each. We looked for consistent behavior across environments, and differed between suites, and vice versa. For example, if scores were high for a suite on all environments, but not high for other suites on all environments, the suite would be marked as a potential cause. This is done because there's some indication that flakiness specific to the suite independent of the environment. We also computed mean, std dev. and the proportion of flaky tests found in each. These were used to interpret distributions and help comparisons.

Results: Figure 3 shows the results. Columns correspond to environments and rows test suites. The x-axis is the score from the W_{FR} model, and each point is approximately one flaky test. The figure indicates the majority of flakiness is coming from a subset of environments and suites. Specifically, large differences can be seen between CI environments despite similar tests running on each. E_{pm} has many more flaky tests, containing nearly 96% of all flaky tests in R_2. Its tests also have higher average flake scores with means of 0.21, 0.1. and 0.08 across all suites (ts_1-ts_3). The most similar environment E_d averages are lower at 0.02, 0.06, 0.04. Such a large increase in flakiness in E_{pm} suggests environment related issues independent of tests – such as network conditions, caching processes, or configured QPS – may exist. We've reported these differences to developers who have confirmed and are implementing fixes.

The scores also suggest suite specific causes may exist. For example, ts_1 in E_w shows flakiness, which differs from the other suites on the environment. In general, due to procedural differences on E_w, most tests on the environment have very few re-runs, and therefore, can't be accurately measured by the model. The existence of these flaky tests suggested unexpected behavior. Developers confirmed there was a bug in the test infrastructure, which has since been fixed.

Other suite related issues may also exist, however, these are still under investigation. For example,ts_2 has a consistent high level of

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

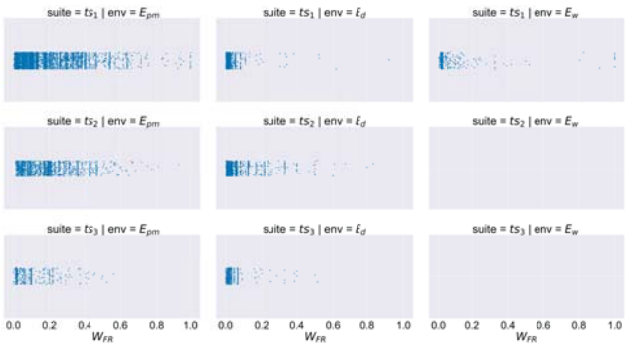


Figure 3: Flakiness across environments and test suites in R_2

flakiness across measurable environments, E_{pm} and E_d. This differs from other suites such as ts_3 where the mean score varies across environments (0.21 on E_{pm} and 0.02 on E_d). We've reported these differences to developers.

In conclusion, to answer RQ3, we confirm the model's scores can be used to identify flakiness causes. The scores detected environment and 2 suite specific issues in R_2. All have been acknowledged by developers with 1 fixed, 1 confirmed and another under investigation. These results help us answer **RQ3**.

5.5 RQ4: Reducing Flakiness

We also evaluated how we can reduce the amount of flakiness seen during development.

Procedure & Metrics: We ran a simulation where results from flaky tests (tests with scores above a defined threshold) were automatically ignored. Specifically, we took the scores at the end of 1 month, and used them to filter results on the succeeding week. All results from tests with scores above a defined threshold τ were ignored.

To determine τ, we ran a grid search for each model type on each environment and searched values between [0, 1] with steps of 0.1. For each iteration of the search, test results were filtered to ignore results from tests with scores above τ. To incorporate the confidence value in the filtering decision, the test's lower bounded score was used. Lower bounds for scores were computed by subtracting the MOE from the test's score, and pushing negative values to 0. MOE was computed with a 95% confidence level from scores over the last 21 days.

Each search looked to optimize two metrics, *flake reduction* (FR) and *loss in fault detection* (LFD), where the objective was to maximize FR and minimize LFD. FR was defined as the percentage of flaky failures removed by filtering, and LFD was the percentage of faults removed. FD was measured by computing the percentage of failures in filtered results that were flaky, where a failure was considered flaky if the test exhibited both pass and fail results for that version. No mapping of failures to faults existed for the dataset, therefore, LFD was estimated by computing the percentage of failures in the filtered set that were considered non-flaky (i.e. the test failed in all re-runs of that version).

We manually analyzed the results of each search to identify the best performing models for each environment. Best performing models were considered to have an optimal balance between high FR and low LFD. Since certain use cases may prioritize FR vs. LFD differently, our best models serve as estimates of the reduction that can achieved.

Finally, to answer RQ4, we report both the total FR and LFD across all environments, as well as the the best performing model on each. The total FR and LFD is defined as above, but calculated over the combined filtered results of best models.

Results: Table 6 shows the results of the filtering simulation run on 1 week of test runs. Columns Env and Model list the best performing models on each environment. FR is the percentage of flakiness removed by filtering, and LFD is the loss in fault detection, where faults are defined as deterministic failures on a version.

Overall, the models reduced flakiness by 26.28% with 2.19% LFD, however, the results varied largely by environment. Best performing models were on E_d and E_w, which reduced flakiness by nearly 50% with less than 1% LFD. Models on E_{pm} were still able to reduce flakiness (25% FR) but had higher LFD (4.46% LFD).

Table 6: Best performing models for filtering flakes.

Env	Model	τ	*FR*	*LFD*
E_{pm}	U_{FR}	0.02	25.02%	4.46%
E_d	W_E	0.05	38.81%	0.58%
E_w	W_{FR}	0.03	82.11%	0.002%
Total (W & D)			44.18%	0.55%
Total			26.28%	2.19%

The difference between environments is partially due to the nature of flakiness on each. We see best results for filtering when flakiness is caused by predictably flaky tests. This was the case with E_d and E_w where flakiness came from a predictable set of tests. E_{pm} was less predictable, and many tests which had flaked in previous weeks did not flake again in the week evaluated. Since the scores in the experiment have a low latency in the experiment (1 week), we are currently evaluating whether improvements could be made on E_{pm} by filtering on more up to date scores of tests.

In summary, the models could be used to reduce flakiness in all environments. The amount of FR and LFD varied based on the nature of flakiness in the tests and CI environment evaluated. Suites and environments where flakiness was caused by more predictably flaky tests saw best results with flakiness nearly halved and less than 1% LFD. Improvements on less predictable flakiness may be possible by improving the freshness of the scores at the time of filtering. These results help us answer **RQ4**.

5.6 RQ5: Detecting Changes in Flakiness Trends

We investigated whether flake scores could be used to identify changes in flakiness trends. Our service is configured to compute scores periodically creating a time series of scores for each test. To test whether the series can be used to detect changes in flakiness over time, we ran the W_{FR} model over the R_3 dataset described in Section 4. R_3 contains a known increase in flakiness reported by developers, which is seen primarily in the ts_3 suite.

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.

Procedure & Metrics: To see if models could detect the change, we processed the dataset over a 5 week window (4 weeks before the increase and 1 week after), simulating updating the scores every 3 days. The resulting time series of average scores were then plotted for each suite, and changes in trends were compared against reports to determine if known changes had been detected.

Results: Figure 4 presents the results. Each line indicates the average W_{FR} score per day for a test suite. Color is used to distinguish the test suites, where orange is ts_1, purple is ts_2, and blue is ts_3. The shaded area on each line is the confidence band for a 95% confidence level. The red dotted line marks the approximate time reports of flakiness began.

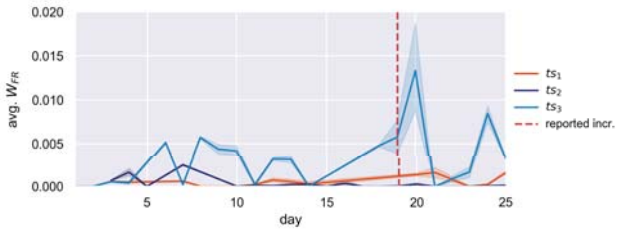


Figure 4: Detecting changes in flakiness trends.

From the figure, we can see the score trends are consistent with reports. The average score increases around the time of reporting primarily for the ts_3 suite. The suite rises by approximately 0.008 from the previous day, which is more than previous rises in flakiness for the suite. ts_2 and ts_1 see little to no rise.

In summary, the model was able to detect an increase in flakiness consistent with developers' reports. We plan to investigate flakiness trends further in future work including trends at the test frame and individual test levels. These results help us answer **RQ5**.

6 CONCLUSION AND FUTURE DIRECTIONS

We have defined a generalizable model for quantifying test flakiness and evaluated the model's accuracy and usefulness on two services at Apple. The model quantifies the flakiness of a test using two measures, flip rate and entropy, and can be configured to give emphasis to a test's most recent behavior. We evaluated the accuracy of the model in two tasks, identification and ranking. We found emphasizing too short of window can give misleading results, and flip rate models performed best in all tasks. We then showed the usefulness of the model in several flake handling tasks including reducing flakiness by filtering and triaging flakiness causes. We concluded by analyzing flakiness trends, and showed our model can detect changes at the aggregated suite level.

Being able to quantify and track the flakiness of a test has many potential uses, which we have only begun to explore in this work. Areas of future work include further trend analysis (including lag), outlier detection, as well as best methods for addressing the cold start problem (i.e., not having a test result history for new tests). Alternative models for quantifying flakiness which leverage additional data beyond an individual test's result history are also worth exploring. The model's scores may also be useful as input to other models. For example, a value quantifying a test's flakiness may be useful for modeling a test's effectiveness, or as an additional signal for test selection.

This research puts forth questions on how flakiness should be quantified. With test flakiness becoming a significant issue and the size of test suites often large in practice, simply identifying flaky tests is not enough. Quantifying test flakiness enables us to better understand the behavior of tests, and better manage flakiness in environments where many tests have some level of flakiness.

ACKNOWLEDGMENT

We thank Nicolas Melo and Hsin-Yin Chou for their support, feedback and manual labeling; Sravya Tumuluru, Donny Lu, Marc-Alexandre Carreira and Karthik Sridhar for providing insights into the service validation process and data; Rujuta Ranade, Maitri Verma, and Praveen Krishnan for providing infrastructure support; and Alex Braunstein, Abhaya Parthy and Sahana Gururaj for their constant support of our work.

REFERENCES

[1] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker. In *ICSE 2018*. 433–444.
[2] Markus Borg, Adnan Causevic, Serge Demeyer, and Sigrid Eldh. 2019. Summary of the 1st IEEE Workshop on the Next Level of Test Automation (NEXTA 2018). *SIGSOFT Softw. Eng. Notes* 43, 4 (Jan. 2019), 53–53. https://doi.org/10.1145/3282517.3282540
[3] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *FSE 2019*. 830–840.
[4] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE 2014* (Hong Kong, China). 235–245.
[5] Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings international parallel and distributed processing symposium*. IEEE, 7–pp.
[6] Zebao Gao. 2017. *Quantifying Flakiness and Minimizing its Effects on Software Testing*. Ph.D. Dissertation.
[7] Zebao Gao, Yalan Liang, Myra B Cohen, Atif M Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE 2015*. 55–65.
[8] Mark Harman and Peter O'Hearn. 2018. 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)*. IEEE, 1–23.
[9] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. 2018. Towards a Bayesian Network Model for Predicting Flaky Automated Tests. In *IEEE International Conference on Software Quality, Reliability and Security*. 100–107.
[10] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-scale Industrial Setting. In *ISSTA 2019*. 101–111.
[11] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *ICST 2019*. 312–322.
[12] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE 2014*. 643–653.
[13] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE 2017*. 233–242.
[14] Atif Memon and John Micco. 2016. *How Flaky Tests in Continuous Integration*. Youtube. https://www.youtube.com/watch?v=CrzpkF1-VsA
[15] John Micco. 2017. The State of Continuous Integration Testing@ Google. (2017).
[16] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSM 2017*. 1–12.
[17] Md Tajmilur Rahman and Peter C Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *FSE 2018*. 857–862.
[18] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA 2014*. 385–396.

Authorized licensed use limited to: Apple. Downloaded on April 16,2025 at 02:15:56 UTC from IEEE Xplore. Restrictions apply.