The Vocabulary of Flaky Tests in Swift

João Medeiros*
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil
jpcm2@cin.ufpe.br

Denini Silva*
Universidade Federal Rural de Pernambuco
Belo Jardim, Pernambuco, Brazil
denini.gabriel@ufrpe.br

Breno Miranda
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil
bafm@cin.ufpe.br

Abstract

Flaky tests produce non-deterministic outcomes without code change, eroding CI confidence and delaying deliveries. While vocabulary-based machine learning prediction has proven effective for Java and JavaScript, no study has evaluated it for Swift, a language whose testing style is dominated by UI and asynchronous code. We collect 91 flaky and 22,349 stable tests from 15 open-source Swift projects via re-execution and commit-history mining, then train five classifiers (*Random Forest*, *Decision Tree*, *Naive Bayes*, *SVM*, *KNN*) on *TF-IDF* unigram+bigram features under stratified 5-fold cross-validation. *Random Forest* achieves the best performance (Precision = 0.92, F1 = 0.86, AUC = 0.95) and substantially outperforms trivial baselines, among them a vocabulary-threshold rule applied to the most informative tokens, confirming a genuine discriminative signal (MCC = 0.75 vs. 0.08 for the best baseline). Information-gain analysis reveals two complementary signal types. *Flakiness markers* appear predominantly in unstable tests and comprise concurrency primitives (*async*, *await*), expectation-based synchronisation (*expectation*, *fulfill*), error propagation (*throws*), and explicit timing dependence (*timeout*, *wait*, *now*). *Stability markers*, chiefly the assertion vocabulary of plainly synchronous tests (*xctassertequal*), count as evidence *against* flakiness. Error analysis shows that the model fails when flakiness is hidden in shared infrastructure outside the test body or when async constructs are used in a deterministic context, exposing the intrinsic limit of lexical prediction. These results extend vocabulary-based flakiness detection to the Swift ecosystem and characterise both its effectiveness and its boundaries.

Keywords

Flaky tests, Unstable tests, Software testing, Flakiness, Swift, iOS

1 Introduction

In software development, tests are fundamental for identifying defects and verifying that fixes work correctly [12]. Ideally, test outcomes remain stable across executions of an unchanged codebase. However, *flaky tests*, that is, tests that non-deterministically pass or fail without any change to the code or the execution environment, are a frequent reality [20]. Their causes are diverse, including concurrency issues, asynchronous operations, test-order dependencies, mismanagement of external resources, dependence on system time, and the use of random values [16], as well as restricted ranges of expected values and *timeout*-related issues [7].

Although flaky tests do not directly affect software functionality, they impose a substantial cost on deployment pipelines [10, 16, 17]. Micco [17] reported that 16% of the 4.2 million tests evaluated in 2017 at Google exhibited unstable behavior, causing recurring delays in *releases*, and that between 2% and 16% of computational capacity was consumed exclusively on re-executing these flaky tests.

Detecting flaky tests is itself challenging. The most common approach, repeated execution of the test suite [4, 14], is costly and time-consuming. This has motivated alternative techniques based on coverage differences across versions, randomization of implementations, injection of stress on computational resources, and static analysis of code features associated with known root causes, including concurrency, asynchrony, and randomness [19].

Among static approaches, machine learning models trained on test-code features have shown promise as predictors of instability [2, 28]. Pinto et al. [20] introduced the notion that flaky tests exhibit a characteristic *vocabulary*, that is, identifiers and keywords that occur disproportionately in them, and showed that classifiers such as *Random Forest* and *SVM* trained on this vocabulary reach an *F-score* of 0.95 for Java. Rampim Soratto and Graciotto Silva [24] replicated this methodology for JavaScript, and Ahmad et al. [1] compared the linguistic diversity of flaky-test vocabularies across Java, Python, C++, Go, and JavaScript using a similar set of classifiers.

Despite this growing body of evidence, the vocabulary-based approach has, to the best of our knowledge, never been evaluated for Swift, the primary language for iOS, macOS, and other Apple-platform development. Swift differs from the ecosystems studied so far in ways that are directly relevant to flakiness: its concurrency model is centered on *async*/*await*, *Grand Central Dispatch*, and actors, and its test suites rely heavily on UI tests written with *XCTest*/*XCUITest*, which interact with rendered interface elements rather than purely programmatic state. Empirical studies of flaky tests in mobile and UI-driven suites report root causes, such as UI-element timing and platform-specific concurrency, that are largely absent from the Java and JavaScript projects studied previously [25, 27]. If the lexical patterns that signal instability in those ecosystems do not transfer to Swift, vocabulary-based tools built around them would offer limited help to iOS developers. This is the gap this work addresses: whether the unstable vocabulary identified for Java and JavaScript generalizes to a language and ecosystem with a markedly different concurrency model and a far stronger reliance on UI testing.

To address this gap, this work investigates the static prediction of *flaky tests* in Swift using machine learning applied to the vocabulary extracted from the source code of the tests. Our approach is organized as a six-stage workflow (Section 4) that (i) collects test cases from open source Swift projects through two complementary sources, namely systematic re-execution of test *suites* and the mining of commit and pull-request histories where developers document and fix instability; (ii) labels each test and reduces it to its discriminative lexical units; and (iii) trains classification models

*These authors contributed equally to this work.

on the resulting vocabulary to predict a test's propensity to be *flaky* without executing it, while extracting an interpretable ranking of the tokens most strongly associated with instability.

We apply this methodology to a dataset of 91 flaky and 22,349 non-flaky tests collected from 15 open source Swift projects. Among the five classifiers evaluated (*Random Forest*, *Decision Tree*, *Naive Bayes*, *SVM*, and *KNN*), *Random Forest* achieved the best performance, with a Precision of 0.92, *F1-Score* of 0.86, and *AUC* of 0.95, with *SVM* statistically indistinguishable from it. Information-gain analysis further shows that the most discriminative tokens divide into two complementary groups. *Flakiness markers* appear predominantly in unstable tests and comprise concurrency primitives (*async*, *await*), expectation-based synchronisation (*expectation*, *fulfill*), error-propagation constructs (*throws*), and explicit timing dependence (*timeout*, *wait*). *Stability markers*, in turn, suppress the flaky prediction; the most robust of them is *xctassertequal*, the assertion vocabulary of plainly synchronous tests.

The main contributions of this work are:

(1) We construct the first dataset of flaky and non-flaky Swift tests, combining systematic re-execution with the mining of commits and pull requests across 15 open source projects.
(2) We evaluate five machine learning classifiers for static, vocabulary-based flaky-test prediction in Swift, showing that an approach validated for Java and JavaScript transfers to a language with a distinct concurrency model and a strong reliance on UI testing.
(3) We identify a *Swift vocabulary of flakiness*, showing that it comprises both *flakiness markers* and previously unreported *stability markers*, and relate both groups to root causes of instability reported for other languages.
(4) We structure the evaluation around four research questions that, beyond predictive accuracy, contextualize the classifiers against trivial baselines and qualitatively analyze representative misclassifications to delineate the limits of vocabulary-based prediction.
(5) We assess the robustness of the hybrid labeling strategy through a validation analysis restricted to tests confirmed as flaky via local re-execution.

2 Motivating Example

To ground the discussion that follows, consider *testShareSheetSendToDevice*, a *flaky* test from the *firefox-ios* project [18] whose repair we recovered from a *pull request* [11]. Listing 1 shows the test before and after that fix. In its flaky form, the test opens the browser's share sheet, waits for the "Send to Device" button to appear, taps it once (line 3, marked −), and then asserts that the expected follow-up screen is reached (lines 10–11). This test is representative of the UI-driven, asynchronous code that dominates Swift/iOS suites, and of the async-assertion patterns our study identifies as the dominant flakiness signal in Section 6.5.

The instability arises from a subtle gap between an element *existing* and being ready to receive input. The call to `.waitAndTap()` blocks only until the button is present in the view hierarchy; it has no way of knowing whether the share sheet has finished its entry animation. Because that animation runs asynchronously, the single tap on line 3 sometimes arrives while the view is still mid-animation, before it has settled into its final, stable position, and the UI layer simply drops the tap, with no error or visible effect. When this happens the screen never advances and the assertions on lines 10–11 fail; on a marginally slower run the tap lands and the very same test passes. Two outcomes for one unchanged revision is the defining symptom of *flakiness* [16], and races of exactly this kind, between a test's actions and the timing of UI rendering, are among the most frequently reported causes of flakiness in mobile and UI-driven suites [7, 25, 27].

Flaky version and fix (PR #27270)

```
1    func testShareSheetSendToDevice() {
2     openNewShareSheet()
3  −  app.staticTexts["Send to Device"].waitAndTap()
3  +  let btn = app.staticTexts["Send to Device"]
4  +  var attempts = 2
5  +  while btn.isVisible() && attempts > 0 {
6  +  btn.waitAndTap()
7  +  waitForNoExistence(btn)
8  +  attempts -= 1
9  +  }
10    waitForElementsToExist([ ... ])
11    ...doneButton.waitAndTap()
12   }
```

Listing 1: Flaky UI test and its fix [11]: the single tap (−) becomes a bounded retry (+).

The developer's fix (Listing 1, lines 3–9) neither inserts a fixed sleep nor speeds the test up; instead, it makes the test *observe* the effect of its own action. The tap is wrapped in a loop that re-taps while the button is still visible and calls `waitForNoExistence` (line 7) to confirm that the button has actually disappeared and the tap therefore took effect, capping the attempts so a genuinely stuck interface cannot hang the suite. An implicit timing assumption is thereby replaced by an explicit synchronization on observable UI state, the canonical repair for this class of flakiness [13, 25].

This example also exposes why detecting such defects is hard in the first place. The two outcomes are separated only by sub-second differences in animation timing, so the failure is not reproducible on demand: confirming it dynamically would mean rerunning the test many times and hoping to land on the unlucky schedule, and in UI-driven suites, where each run drives a full application through its interface, that cost is multiplied by long execution times and large input spaces [25]. The very property that makes this flakiness expensive to trigger, its dependence on runtime timing, also makes it expensive to confirm by running the test.

Yet nothing about the underlying cause is hidden at rest. The entire episode, both the defect and its repair, is legible *statically*: the test is saturated with UI locators (*staticTexts*, *accessibilityidentifiers*) and explicit synchronization primitives (*waitAndTap*, *waitForNoExistence*), and the fix does nothing but add more of the same vocabulary. The timing assumption that caused the flakiness, and the synchronization that removed it, are both written into the source.

This is the tension our work is built on: a fault whose *behaviour* only manifests at runtime nonetheless leaves a stable *lexical* trace in the test code, one that is available without executing the suite.

That trace, however, is expressed through constructs specific to Swift and *XCUITest*, an ecosystem on which the vocabulary-based approach to flakiness has never been evaluated (Section 3). Whether the iOS-flavoured vocabulary illustrated here is idiosyncratic to this one test or a recurring, learnable signal across real Swift projects is precisely what motivates the study that follows.

3 Related Work

Flaky tests. A *flaky test* produces different outcomes across executions of the same version of the software, without any change to the implementation [16]. Its cost is well documented: developers spend considerable effort triaging failures that signal no real defect [20], and repeated exposure erodes trust to the point that genuine failures are ignored, harming software stability [21]; surveys of practitioners confirm that flakiness is a recurring, everyday concern rather than an occasional nuisance [9].

Detecting flaky tests. Detection techniques are commonly classified as dynamic, hybrid, or static, according to whether the test must be executed [20, 24]. The dynamic approach simply reruns the suite until a test disagrees with itself [19], a strategy deployed at scale at Google and Microsoft [13, 17]; its cost, however, grows with the rerun budget, and even five reruns surface only about 88% of *flaky* tests [15]. Hybrid techniques reduce this cost by pairing a single execution with additional analysis: *DeFlaker* flags a failing test as *flaky* when it does not exercise recently modified code, matching the detection rate of repeated re-execution at a fraction of the overhead [4], while *Shaker* injects CPU and memory stress across test runs to surface concurrency-related *flakiness* more quickly [26]. Even so, hybrid methods still require executing the suite at least once, which remains prohibitive for large projects.

Vocabulary-based prediction. This cost motivates *static* prediction, which infers *flakiness* from source code alone. Pinto et al. [20] observed that *flaky* tests share a characteristic *vocabulary*, identifiers and keywords that recur disproportionately, and trained classifiers such as *Random Forest* and *SVM* that reached an F1-score of 0.95 on a Java dataset, with the most informative tokens (*job*, *table*, *id*, *action*) pointing to root causes such as remote task execution and event queues. Subsequent work showed the signal to be robust across feature sets and classifiers, whether combining lexical with behavioral features (*FlakeFlagger* [2]) or applying nearest-neighbor search over code vectors (*FLAST* [28]), and replications extended it beyond Java to JavaScript [24] and across five languages including Python, C++, and Go [1]. Barbosa et al. [3] further show, via re-execution across languages, that root causes are only partially shared and vary by ecosystem, evidence that language matters. Crucially, none of these studies covers Swift or any language with a UI-test-dominated suite.

Learned features. A parallel line of work replaces hand-crafted vocabulary with pretrained language models, ranging from fine-tuning CodeBERT on test code, with reported F1-scores between 0.79 and 0.98 [8], to prompting general-purpose LLMs, which in a recent evaluation performed only marginally above random guessing on test code alone [5]. The evidence here is mixed: learned features help most for root-cause categorization and far less for the binary flaky/non-flaky prediction addressed in this work [22], and once class imbalance is handled realistically their reported accuracy drops sharply (F1 from 0.82 to 0.57), with models leaning on superficial tokens rather than code semantics [23]. Vocabulary-based features, by contrast, remain computationally cheap and directly interpretable, supporting the kind of root-cause analysis pursued in this work.

Mobile and UI testing. Finally, studies of mobile and UI-driven suites, the dominant testing style in the Swift/iOS ecosystem, report root causes, and presumably vocabularies, that differ from those of JVM and web code: Thorve et al. [27] identify mobile-specific causes in Android not previously reported for non-mobile software, and Romano et al. [25], analyzing 235 *flaky* UI tests from 62 web and Android projects, show that the large input spaces of UI tests make repeated re-execution especially impractical, reinforcing the case for static detection. Swift/iOS, with its *async*/*await* concurrency model and heavy reliance on *XCTest*/*XCUITest*, sits squarely in this gap: it has a markedly different concurrency model and a far stronger UI-testing focus than any ecosystem on which vocabulary-based prediction has been evaluated so far. This work closes that gap, motivating the research questions investigated next.

4 Approach

Our approach builds an interpretable *Swift vocabulary of flaky tests* and uses it to predict flaky tests directly from source code. It rests on a single premise: although flakiness manifests only at runtime, it leaves a stable *lexical* trace in the test code.

To operationalize this premise we first construct a labeled dataset from 15 open-source Swift projects (see Table 1) using two complementary sources: *local re-execution* and *history mining*; all remaining tests constitute the stable pool. Figure 1 summarizes the resulting six-step workflow. We describe each step below, stressing the rationale behind each decision.

Step 1: Test Sources. Ground truth for flakiness is hard to obtain because no single oracle is at once precise and complete. We therefore draw candidate tests from open-source Swift projects (Table 1) through two complementary sources, *local re-execution* and the *mining of commit and pull-request histories*. The two are deliberately combined because their precision/recall trade-offs are complementary: re-execution yields high-confidence labels but only for non-determinism that reproduces in our environment, whereas mining captures a far broader range of developer-diagnosed root causes, so their union forms a more representative flaky set than either source alone.

Step 2: Test Selection. Each source is then turned into flaky and stable labels. By re-execution, we ran each test 50 times against an unchanged revision on a local macOS machine (Apple M1 Pro, 16 GB RAM); a test that produced at least one disagreeing outcome is labeled flaky [16]. By mining, we collect commits and pull requests referencing "*flaky*" and keep only those that genuinely repair a flaky test, confirmed by manual inspection of the diff to verify that the changed method is a test and that the commit message or PR description attributes the fix to non-determinism. We then recover the test at the parent of the fixing commit, since that pre-fix

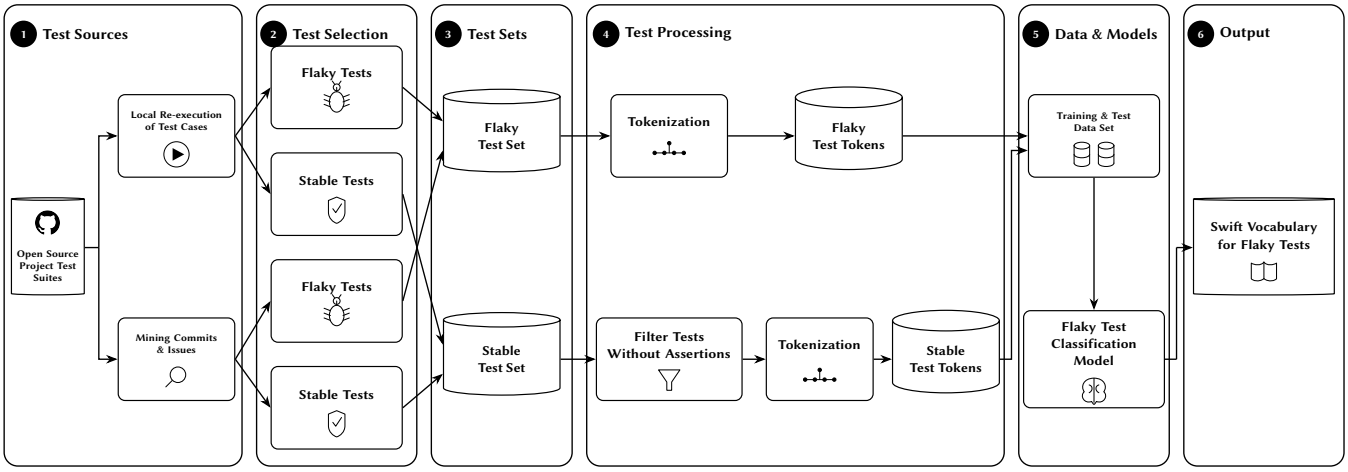

Figure 1: End-to-end workflow for constructing the Swift vocabulary of flaky tests.

version is the artifact that still exhibited the behavior (e.g., firefox-ios PR #27270 [11], which fixes two flaky UI tests). Tests flagged by neither source constitute the stable pool.

Step 3: Test Set Construction. The flaky tests confirmed by the two sources are merged into a single *flaky test set* and the remainder into a *stable test set*. Unifying the sources is what lets one classifier learn from the full diversity of observed instability, regardless of how each case was discovered; the resulting composition and its pronounced class imbalance are detailed in Section 5, and we assess the soundness of this hybrid labeling in Section 6.3.

Step 4: Test Processing. Each test is reduced to the lexical units that carry the signal. We first discard stable tests with no *XCTest* assertion, as a method that verifies nothing is not a genuine test and would only blur the contrast between classes, and then tokenize the rest after stripping line and block comments. We then remove 23 structural Swift keywords, spanning declarations (*let*, *func*, *class*, …) and control flow (*if*, *for*, *guard*, …); being near-ubiquitous, they would only crowd out the discriminative vocabulary. Keywords tied to concurrency or error propagation (*async*, *await*, *throws*, *try*, *defer*) are deliberately retained, and the full list is in the artifact. The output is two token collections, one per class.

Step 5: Model Construction. Tests are vectorized with *TF-IDF* over unigrams *and* bigrams, since several instability cues are meaningful only as adjacent pairs (e.g., *async throws*, *load ordering*); the vectoriser keeps terms occurring in at least two documents and in at most 90% of them, with sublinear term-frequency scaling. To keep our conclusions independent of any single learner, we train and compare five classifiers (*Random Forest*, *Decision Tree*, *Naive Bayes*, *SVM*, *KNN*) [20, 24]. *Random Forest* uses 200 trees and *SVM* a linear kernel; both, with *Decision Tree*, use balanced class weights, while *Naive Bayes* and *KNN* keep scikit-learn defaults. The five span distinct inductive biases and match those of the studies we build on, making comparison direct; with only 91 positives, fine-tuning a language model would risk overfitting and forgo the interpretability Section 6.5 depends on. Because the corpus is severely imbalanced, we undersample the majority class and evaluate under stratified 5-fold cross-validation.

Step 6: Vocabulary Extraction. Finally, we rank tokens by information gain with respect to the flaky/stable label, obtaining the *Swift vocabulary of flaky tests*: the lexical units whose presence most reduces uncertainty about a test's class. This ranking is what makes the approach explanatory rather than opaque, letting the predictive signal be traced to concrete constructs and mapped onto known root causes of flakiness.

5 Objects of Analysis

The unit of analysis in this study is the individual Swift *test case*, the smallest artifact for which a flaky/stable label is meaningful and the granularity at which the vocabulary signal is learned. We collected these test cases from the 15 open-source Swift projects listed in Table 1. The projects span the breadth of the Swift ecosystem relevant to our research questions: networking and server-side frameworks (*Alamofire*, *swift-nio*, *vapor*), data and persistence libraries (*GRDB.swift*, *Kingfisher*, *Nuke*), full iOS/macOS applications with substantial UI test suites (*firefox-ios*, *DuckDuckGo iOS*, *Signal-iOS*, *Maccy*), and core developer tooling (*swift-package-manager*, *swift-composable-architecture*, *swift-snapshot-testing*, *sentry-cocoa*). Such breadth suits our questions, exercising both the *async*/*await* concurrency model and the UI-testing style that distinguish Swift from the ecosystems studied in prior work.

Applying the labeling procedure of Section 4 to these projects yields **91 flaky** and **22,349 non-flaky** test cases. Two properties of this dataset shape the evaluation that follows. First, the distribution is *severely imbalanced*, with flaky tests accounting for roughly 0.4% of all tests; this mirrors the reality of production suites and directly motivates both the undersampling strategy of Step 5 and our use of imbalance-aware metrics. Second, as shown in Table 1, 66 of the 91 flaky tests were obtained through history mining and 25 through local re-execution, with some projects contributing flaky tests from both sources. This hybrid composition lets us study whether the labeling strategy is sound by re-running the analysis on the re-execution subset alone.

Table 1: Subject projects and dataset composition. Flaky tests are split by labeling source: *Mined* denotes tests recovered from commit/PR histories; *Re-exec.* denotes tests identified by running each test 50 times.

		Flaky Tests			
Project	★	**Mined**	**Re-exec.**	**Total**	**# Non-Flaky**
Alamofire	42.4k	1	2	3	554
DuckDuckGo iOS	1.9k	0	17	17	1003
firefox-ios	13k	8	1	9	4487
GRDB.swift	8.5k	9	0	9	172
ios-oss	8.7k	1	0	1	1235
Kingfisher	24.3k	2	1	3	703
Maccy	20.3k	2	0	2	32
Nuke	8.6k	11	0	11	126
sentry-cocoa	1.1k	3	0	3	5041
Signal-iOS	12.1k	7	0	7	713
swift-composable-architecture	14.7k	2	0	2	686
swift-package-manager	10.2k	9	0	9	1173
swift-nio	8.5k	10	0	10	2657
swift-snapshot-testing	4.3k	0	2	2	–*
vapor	26.1k	1	2	3	3767
Total	–	**66**	**25**	**91**	**22,349**

* Suite not covered by the stable-test extraction.

6 Evaluation

We evaluate whether vocabulary-based prediction of flaky tests is effective in the Swift ecosystem, whether its results reflect a genuine lexical signal, and what that signal reveals about Swift-specific root causes of flakiness. We structure this investigation around four research questions, the first of which comprises two sub-questions addressing predictive effectiveness and labeling robustness.

RQ1.1 — Predictive Effectiveness. With what accuracy can flaky test cases be predicted from the vocabulary (identifiers and keywords) present in the source code of Swift test cases?

Rationale. Vocabulary-based prediction has been shown to work for Java and JavaScript, but never for a language with the *async*/*await* concurrency model and UI-test focus of Swift. RQ1.1 establishes whether the lexical trace of flakiness is strong enough, in this ecosystem, for standard classifiers to separate flaky from stable tests using static features alone.

RQ1.2 — Robustness of the Hybrid Labeling Strategy. Does the predictive performance achieved by vocabulary-based classifiers remain stable when flaky tests are labeled exclusively through local re-execution, without the commit-history mining component of the hybrid strategy?

Rationale. The conclusions of RQ1.1 are only meaningful if the flaky labels are reliable. Most labels come from commit and pull-request mining, which relies on developers explicitly documenting non-determinism and may admit noise. RQ1.2 stress-tests the labeling strategy by replicating the RQ1.1 evaluation exclusively on the 25 tests whose flakiness was confirmed by local re-execution, where the label is unambiguous.

RQ2 — Comparison with Trivial Baselines. Does vocabulary-based prediction substantially outperform trivial baseline classifiers that disregard the test vocabulary?

Rationale. On a balanced two-class problem, even naive predictors achieve non-zero scores, so high absolute metrics do not, by themselves, demonstrate that the model has learned a meaningful signal. RQ2 contextualizes the results of RQ1.1 against trivial baselines, ensuring that the observed performance reflects genuine discriminative power rather than an artifact of the evaluation setup.

RQ3 — The Vocabulary of Flaky Swift Tests. Which identifiers present in Swift test code are most strongly associated with test instability?

Rationale. Beyond predictive accuracy, a central goal of this work is interpretability. RQ3 asks which lexical units drive the prediction, so that the signal can be traced to concrete Swift/iOS constructs and mapped onto known root causes of flakiness, and compared with the vocabularies reported for other languages.

RQ4 — Error Analysis. When and why does the vocabulary-based classifier mispredict, and what do these errors reveal about the limits of lexical prediction?

Rationale. Aggregate metrics hide the conditions under which a vocabulary-based model breaks down. RQ4 examines representative misclassifications qualitatively, to understand the limits of lexical prediction and to inform when a static, vocabulary-based detector should and should not be trusted.

6.1 Experimental Setup

All research questions are evaluated under the protocol of Step 5 (Section 4): stratified 5-fold cross-validation over class-balanced data obtained by random undersampling of the majority class. Three properties determine how the numbers should be read.

First, undersampling is applied to the corpus as a whole, so the *test* folds are balanced at roughly 1:1 rather than left at the natural 0.4% rate. Every Precision reported here is therefore measured on an artificial class ratio, not the one a detector would face in production; MCC and AUC are the metrics to compare across settings. Second, a single undersample is not a stable basis for reporting, since which 91 stable tests are drawn moves F1 by several points on its own: we repeat the draw 30 times and report the mean over the resulting 150 fold-level estimates, with a 95% confidence interval. Third, *TF-IDF* is fitted on the training fold alone, so no information from held-out tests reaches the feature space, and information gain serves only interpretation (RQ3) and the vocabulary-threshold baseline (RQ2), never feature selection: training always uses the full *TF-IDF* space.

To avoid the well-known pitfalls of accuracy on imbalanced data, we report five complementary metrics, namely *Precision*, *Recall*, *F1-Score*, the *Matthews correlation coefficient* (MCC), and the area under the ROC curve (AUC), placing particular emphasis on F1, MCC, and AUC, which remain informative when the classes are unequally represented and which prior vocabulary-based studies adopt as primary metrics [20, 24]. Confusion matrices are additionally reported per classifier to expose the balance between false positives and false negatives.

6.2 Answering RQ1.1: Predictive Effectiveness

Under the protocol of Section 6.1, all five classifiers separate flaky from non-flaky tests well (Table 2). *Random Forest* and *SVM* lead and are statistically indistinguishable from one another: *Random Forest* attains the highest Precision (0.92), MCC (0.75) and AUC (0.95), while *SVM* attains a marginally higher F1 (0.87 against 0.86), a difference well inside the confidence intervals of both. *Naive Bayes*

Table 2: Classifier results, averaged over 30 draws × 5 folds; 95% CI at most ±0.02.

Algorithm	Precision	Recall	F1	MCC	AUC
Random Forest	0.92	0.82	0.86	0.75	0.95
Decision Tree	0.82	0.74	0.77	0.59	0.79
Naive Bayes	0.81	0.91	0.85	0.70	0.94
Support Vector Machine	0.86	0.89	0.87	0.74	0.94
K-Nearest Neighbours	0.80	0.86	0.82	0.65	0.89

is competitive (F1 0.85) and buys the highest Recall of the five (0.91) at the cost of Precision, *KNN* follows (F1 0.82), and *Decision Tree* trails (F1 0.77). That learners with such different inductive biases agree indicates the signal is carried by the vocabulary itself rather than by any one model: Swift's concurrency idioms (*async*/*await*, *GCD* dispatch, *XCTestExpectation*) are lexically concentrated and recur across flaky tests, yielding a feature space separable even for simple classifiers.

These results indicate that the models can predict *flaky tests* with confidence from attributes extracted statically from the source code. The average confusion matrices (Figure 2) show the trade-off each learner strikes: *Random Forest* is the most conservative, averaging only 1.5 false positives per fold against 3.2 false negatives, whereas *Naive Bayes* inverts that balance (4.1 against 1.6) and *SVM* sits between the two. Which of these is preferable depends on the cost of a missed flaky test relative to that of a spurious warning. These findings are in agreement with [20], which demonstrated that *Random Forest* and *SVM* tend to present greater robustness in flaky-test detection scenarios.

RQ$_{1.1}$: All five classifiers reliably distinguish flaky from non-flaky Swift tests using only static vocabulary features. *Random Forest* achieved the best overall performance (Precision = 0.92, F1 = 0.86, MCC = 0.75, AUC = 0.95), with *SVM* statistically indistinguishable from it.

6.3 Answering RQ1.2: Strategy Robustness

The results of RQ1.1 rest on a hybrid dataset in which 66 of 91 flaky tests were sourced from commit and pull-request mining and only 25 from local re-execution. The commit-mined labels depend on developers explicitly describing a fix as resolving non-determinism, which may admit noise: a commit that references "flaky" might fix a dependency rather than true non-determinism, or a flaky test might be retired rather than repaired. To assess whether this affects the conclusions of RQ1.1, we replicate the full evaluation using only the 25 re-execution-confirmed flaky tests, for which the label is unambiguous. The metrics obtained are presented in Table 3. Every classifier stays within four percentage points of its RQ1.1 F1, and none degrades in a way that would call the labels into question: *Random Forest* moves from 0.86 to 0.84, *SVM* from 0.87 to 0.86, *Naive Bayes* is unchanged at 0.85, *KNN* improves from 0.82 to 0.81, and *Decision Tree* from 0.77 to 0.81. *Random Forest* becomes markedly more conservative on the smaller subset, trading Recall (0.76 against 0.82) for near-perfect Precision (0.97) and the highest AUC observed anywhere in this study (0.98).

Table 3: Classifier results on the re-execution subset.

Algorithm	Precision	Recall	F1	MCC	AUC
Random Forest	0.97	0.76	0.84	0.76	0.98
Decision Tree	0.88	0.79	0.81	0.67	0.82
Support Vector Machine	0.88	0.87	0.86	0.75	0.95
Naive Bayes	0.84	0.87	0.85	0.71	0.94
K-Nearest Neighbours	0.81	0.83	0.81	0.63	0.90

Table 4: Comparison of the best vocabulary-based classifier against trivial baselines, under the same evaluation protocol.

Classifier	Precision	Recall	F1	MCC	AUC
Majority-Class Predictor	0.10	0.20	0.13	0.00	0.50
Random Predictor	0.48	0.49	0.49	−0.04	0.48
Vocab. Threshold (top-10)	0.54	0.81	0.64	0.08	0.52
Random Forest	0.92	0.82	0.86	0.75	0.95

These results support the robustness of the hybrid approach. If the commit-mined labels were substantially noisy, restricting the positive class to the 25 unambiguous re-execution cases should have *improved* the metrics sharply, and training on the noisy superset should have degraded them; neither happens. The signal learned from the mined tests is therefore consistent with the signal learned from the re-executed ones. What the subset does cost is coverage: with 25 positives the confidence intervals roughly double (up to ±0.04), which is why we base the main analysis on the full set.

RQ$_{1.2}$: All five classifiers are robust to the labeling strategy, staying within four percentage points of their RQ1.1 F1 when the positive class is restricted to the 25 re-execution-confirmed tests (*Random Forest*: 0.84; *SVM*: 0.86; *Naive Bayes*: 0.85; *Decision Tree* and *KNN*: 0.81). Restricting to unambiguous labels neither improves nor degrades the results appreciably, which is what one expects if the commit-mined labels are sound.

6.4 Answering RQ2: Trivial Baselines

We compare the best-performing classifier against three trivial baselines under the same protocol: a *Majority-Class Predictor* (always predicts the dominant class), a *Random Predictor* (draws from the class distribution), and a *Vocabulary Threshold* (predicts flaky if the test contains any of the top-10 tokens by mutual information on the training fold). The first two measure what is attainable without any vocabulary signal; the third isolates the contribution of *learning* over a simple presence rule. Table 4 reports the comparison.

The three baselines reveal complementary failure modes. Because undersampling makes the two classes exactly equal in size, the Majority-Class Predictor is degenerate: it has no majority to fall back on and its label depends on how the tie is broken in each fold, which is why its Precision, Recall and F1 are unstable and meaningless. Its MCC of 0.00 and AUC of 0.50 are the informative entries, and they confirm that no discriminative information flows from the label distribution alone. The Random Predictor adds label variation but likewise stays at chance (MCC = −0.04, AUC = 0.48).

Random Forest

	P:NF	P:F
A:NF	**16.7**	1.5
A:F	3.2	**15.0**

Decision Tree

	P:NF	P:F
A:NF	**15.2**	3.0
A:F	4.7	**13.5**

KNN

	P:NF	P:F
A:NF	**14.1**	4.1
A:F	2.6	**15.6**

Naive Bayes

	P:NF	P:F
A:NF	**14.1**	4.1
A:F	1.6	**16.6**

SVM

	P:NF	P:F
A:NF	**15.4**	2.8
A:F	1.9	**16.3**

Figure 2: Mean confusion matrices per fold over 30 draws × 5 folds. P = Predicted, A = Actual, NF = Non-Flaky, F = Flaky.

The Vocabulary Threshold is the most diagnostic baseline: by construction, it exploits the exact tokens identified in RQ3 as most informative, the same tokens the learned model uses as features, yet it achieves an MCC of only 0.08 and an AUC of 0.52, barely above chance. The high Recall (0.81) paired with near-random Precision (0.54) reveals why. The tokens most associated with flakiness are frequent enough in the stable class that mere presence carries almost no information: a rule that flags any test containing one of them ends up flagging most of the corpus. Knowing that a test *contains* a "flaky" token is insufficient; what matters is the weighted co-occurrence profile across the full vocabulary, which is precisely what TF-IDF combined with Random Forest learns. Raising the threshold to the top-20 tokens does not help (MCC 0.09), confirming that the limitation is the presence rule itself rather than the size of the token set.

Against this backdrop, Random Forest's MCC of 0.75 and AUC of 0.95 stand well clear of every baseline, improving MCC roughly ninefold and AUC by 0.43 points: the model does not merely detect the presence of flaky vocabulary; it learns the discriminative pattern *within* that vocabulary.

RQ$_2$: Random Forest substantially outperforms all trivial baselines, including a Vocabulary Threshold that directly exploits the top-10 mutual-information tokens yet achieves only MCC = 0.08. The gap confirms that the model captures a genuine lexical signal that token presence alone cannot replicate.

6.5 Answering RQ3: Flakiness in Swift

Table 5 presents the twenty tokens with the highest information gain together with their *document frequency* in each class: the percentage of tests in which the token appears at least once. Document frequencies are essential context because information gain measures discriminative power in *either direction*, a token is highly informative whether it appears mostly in flaky tests (*flakiness marker*) or mostly in stable tests (*stability marker*, denoted †). Distinguishing these two roles is critical for correct interpretation and is a finding not reported by prior vocabulary studies for other languages.

Flakiness markers. The tokens that appear predominantly or almost exclusively in flaky tests are the strongest positive predictors of instability. *Async* (38% flaky, 1% stable) and *await* (30% vs. 1%) are the clearest examples: each is present in roughly one in three flaky tests and in almost none of the stable ones, making them the single strongest lexical signal. Their meaning is direct: tests that call *async*/*await* functions without sufficient synchronisation control race against the scheduler and produce non-deterministic outcomes. The bigram *async throws* (14% vs. 0%) sharpens the same signal, picking out asynchronous calls that can also fail.

A second group is associated with *error propagation and assertion patterns in asynchronous code. Throws* (40% vs. 1%) appears in two

Table 5: Top-20 tokens by information gain (*IG*), averaged over the 30 undersampling draws of RQ1.1. %F and %S: share of flaky and stable tests containing the token at least once.

Token	IG	%F	%S	Token	IG	%F	%S
xctassertequal†	0.193	60	80	*relaxed*†	0.083	1	25
async	0.142	38	1	*ordering relaxed*†	0.078	1	25
throws	0.137	40	1	*try*	0.071	44	17
await	0.109	30	1	*expectation*	0.070	29	4
destroy†	0.105	0	26	*wait*	0.070	34	6
timeout	0.104	42	5	*description*	0.069	32	6
xctassertequal load†	0.102	0	26	*now*	0.065	20	2
unsafeatomic†	0.098	0	26	*ordering*†	0.064	2	26
defer destroy†	0.091	0	26	*fulfill*	0.062	27	5
load ordering†	0.084	2	26	*async throws*	0.060	14	0

† Stability marker: %S > %F.

out of five flaky tests and almost never in stable ones, indicating tests whose correctness depends on error-throwing async calls that can silently mis-sequence, and *try* (44% vs. 17%) accompanies it at the call site. *Expectation* (29% vs. 4%) and *fulfill* (27% vs. 5%) represent *XCTestExpectation*-based synchronisation, a mechanism specifically designed to wait for asynchronous operations: their prevalence in flaky tests indicates that expectation-based synchronisation, when misconfigured, is itself a source of non-determinism.

A third group is explicitly temporal. *Timeout* (42% vs. 5%) and *wait* (34% vs. 6%) mark the bounded waits through which a test states how long it is prepared to tolerate an asynchronous operation, and *now* (20% vs. 2%) marks a dependence on the current clock. Together they describe tests whose outcome is contingent on execution timing, one of the most frequently reported root causes of flakiness [7, 16].

Stability markers. A key finding of this analysis, one not reported by prior vocabulary studies, is the presence of *stability markers*: tokens that appear more often in stable tests than in flaky ones and whose presence the model uses as evidence *against* flakiness. Half of the top-20 falls into this category, and it divides into two groups of quite different character.

The first is a tightly coupled family of atomics vocabulary: *unsafeatomic*, *destroy*, *ordering*, *relaxed* and the bigrams built from them, each appearing in about a quarter of the stable tests and in essentially none of the flaky ones. These are the lexical signature of Swift's `UnsafeAtomic` API: *ordering* and *relaxed* name the memory-ordering qualifier passed to an atomic operation, and *destroy* the paired deallocation. Tests that manipulate shared state through this API declare their synchronisation explicitly at every access, and in our corpus they are uniformly stable. The signal is strong but narrow, and Section 8 examines how far it generalises.

The second group is broader. *Xctassertequal* is the single most informative token in the whole ranking (IG 0.193) and it is a stability

marker: it appears in 60% of flaky tests but 80% of stable ones. Although present in the majority of *both* classes, it is more prevalent in stable tests, so the model uses it as mild evidence against flakiness. The underlying pattern is that tests saturated with straightforward equality assertions but no concurrent or asynchronous primitives are predominantly stable. It is not the assertion itself that signals instability, but the *absence* of such assertions in tests that instead rely on *expectation* and *fulfill* for asynchronous verification.

Category structure. The top-20 tokens in this dataset organise into four groups: (i) *concurrency and asynchrony* (*async*, *await*, *async throws*, *expectation*, *fulfill*), the dominant flakiness signal; (ii) *async error propagation* (*throws*, *try*); (iii) *timing dependence* (*timeout*, *wait*, *now*), patterns specific to testing asynchronous code; and (iv) *stability markers*, comprising the assertion vocabulary of straightforwardly synchronous tests (*xctassertequal*) and the explicit memory-ordering vocabulary of atomic operations (*unsafeatomic*, *ordering*, *relaxed*). UI interaction tokens (e.g., *accessibilityidentifiers*), while present in the dataset, do not appear among the top-20 discriminative features; this is consistent with the project composition, where 60% of the flaky tests (55 of 91) originate from projects related to networking/server-side, persistence, and developer tooling, rather than UI-heavy applications, so concurrency vocabulary dominates the signal.

RQ$_3$: The top-20 tokens divide into two kinds. *Flakiness markers*, which appear predominantly in flaky tests, comprise concurrency primitives (*async*, *await*, *async throws*), expectation-based synchronisation (*expectation*, *fulfill*), error propagation (*throws*, *try*), and explicit timing dependence (*timeout*, *wait*, *now*). *Stability markers* are evidence *against* flakiness: chiefly *xctassertequal*, the assertion vocabulary of plainly synchronous tests, alongside the explicit memory-ordering vocabulary of atomic operations. This bidirectional discriminative structure reflects both the constructs that introduce non-determinism and those that accompany its absence.

6.6 Answering RQ4: Error Analysis

The aggregate metrics of RQ1.1 and the vocabulary of RQ3 establish *that* lexical prediction works and *which* tokens drive it, but they do not reveal *when* it fails. To understand the limits of vocabulary-based prediction, we conduct a qualitative analysis of the misclassifications produced by the best model (*Random Forest*). A complete cross-validation run produces, on average, 16.0 false negatives and 7.4 false positives. Because any single undersample yields a slightly different error set, we do not read individual errors off one run: instead we record, for every test, the fraction of the 150 models (30 draws × 5 folds) that flag it, and treat a test as a systematic error only when the majority of models that never saw it during training get it wrong. Thirteen of the 91 flaky tests are missed under this criterion. We group the resulting errors into three recurring patterns that together delimit where a purely lexical signal is and is not sufficient.

False negatives: flaky tests without common flaky vocabulary. Some genuinely flaky tests are missed because their source does not contain the tokens most associated with instability (Table 5). These are cases where the non-determinism originates outside the lexically visible test body, for example in a shared helper, a fixture, or an implicit ordering dependency, leaving no characteristic vocabulary for the model to latch onto. A concrete example is `testBuildParametersWithInvalidDevice` from Signal-iOS, which only 3% of models flag. The test exercises cryptographic key-pair generation and mock-based message dispatch; its visible vocabulary consists of domain-specific identifiers (*pniKeyPair*, *localSignedPreKey*, *localRegistrationId*) with no occurrence of the concurrency or timing tokens listed in Table 5. The non-determinism resides in shared mock state coordinated across the test class, outside the test body, a root cause structurally invisible to a lexical classifier. A second example, also never flagged, is shown in Figure 3(a): `testFlush_BlocksCallingThread_TimesOut` from SentryCocoa [6] asserts that a blocking call completes within a ±0.1 s window, yet its timing tokens (*getAbsoluteTime*, *toTimeInterval*) do not appear among the top-20 mutual-information features and are therefore unrecognized as a flakiness signal. These misses are systematic rather than incidental: six of the thirteen are flagged by no model at all.

False positives: stable tests with flaky-associated vocabulary. Conversely, some stable tests are flagged as flaky because they contain tokens that are strongly predictive in aggregate yet, in context, are benign. Figure 3(b) shows the clearest example: `testDownsamplingHandleScale2x` from Kingfisher, flagged by 80% of the models that never saw it. Its lexical profile is that of a textbook flaky test: it opens an *XCTestExpectation*, issues an image retrieval whose result arrives in a closure, and blocks on *waitForExpectations* with an explicit one-second timeout, a structure indistinguishable from tests that genuinely race against asynchronous image decoding. What makes it deterministic is the single call to *stub*, which replaces the network with canned data so the callback fires synchronously inside the test process. The construct that guarantees stability is therefore present in the source, but it is one token among many and carries none of the discriminative weight that *expectation* and *waitForExpectations* do.

Notably, two constructs we expected to mislead the model do not. Tests built on a virtual clock (`testRunScheduler`, which uses *DispatchQueue.test*) or on a synchronous in-process event loop (`testAndAllCompleteWithPreFailedFutures`, which uses *EmbeddedEventLoop*) are flagged by only 7% and 12% of models respectively, and are thus classified correctly. Their deterministic testing infrastructure evidently brings its own vocabulary, which the model learns to read as evidence *against* flakiness, in the same way as the stability markers of Section 6.5.

Borderline cases. Between the systematic errors and the confident predictions lies a band of tests the model cannot settle. *SwiftNIO_testMetricsDelegateTickInfo* and *nuke_testIsLoadingUpdated* are flagged by 43% and 40% of models respectively: genuinely flaky tests whose vocabulary straddles both classes, so the verdict flips with the training sample. This band is the practical reason for reporting the flag rate rather than a single label, and it is where a static detector is least useful on its own.

Limits of vocabulary-based prediction. These errors share one root: vocabulary captures the *constructs* correlated with flakiness but not the *correctness of their use*. The detector is thus most reliable when the test body itself instantiates the primitives that cause

(a) False negative — *testFlush_BlocksCallingThread_TimesOut* [6]

```swift
func testFlush_BlocksCallingThread_TimesOut() {
    givenCachedEvents(amount: 30)
    fixture.requestManager.responseDelay = fixture.flushTimeout + 0.2
    let before = SentryDate.getAbsoluteTime()
    let result = sut.flush(fixture.flushTimeout)
    let elapsed = getDurationNs(before, SentryDate.getAbsoluteTime()).
      toTimeInterval()
    XCTAssertGreaterThan(elapsed, fixture.flushTimeout)
    XCTAssertLessThan(elapsed, fixture.flushTimeout + 0.1) // tight 0.1s
      window
    XCTAssertEqual(.timedOut, result)
}
```

(b) False positive — *testDownsamplingHandleScale2x* (Kingfisher)

```swift
func testDownsamplingHandleScale2x() {
    let exp = expectation(description: #function)
    let url = testURLs[0]
    stub(url, data: testImageData)  // network stubbed
    _ = manager.retrieveImage(with: .network(url),
        options: [.processor(...), .scaleFactor(2)]) { result in
        let image = result.value?.image
        XCTAssertEqual(image?.size, .init(width: 4, height: 4))
        exp.fulfill()   // fires synchronously
    }
    waitForExpectations(timeout: 1, handler: nil)
}
```

Figure 3: Systematic misclassifications: (a) a flaky test flagged by no model; (b) a stable test flagged by 80% of models.

non-determinism (raw async/await, real network calls, shared mutable state), and least reliable when asynchrony is delegated to infrastructure whose determinism its own tokens do not reveal. The natural complements are data-flow analysis, to check whether async primitives are guarded, and cross-test dependency analysis, to surface shared fixtures.

RQ$_4$: Vocabulary-based prediction fails when flakiness is hidden in shared infrastructure outside the test body (false negatives) or when async and timing constructs are used in a deterministic context (virtual schedulers, embedded event loops) that is lexically indistinguishable from a real one (false positives). In both cases, the model detects the relevant constructs but cannot assess the correctness of their use, a distinction that requires semantic rather than lexical analysis.

7 Discussion

The answers to RQ1.1, RQ1.2, and RQ3 are complementary: RQ1.1 shows that the vocabulary of a Swift test, captured through simple TF-IDF features, is sufficient for classifiers such as *Random Forest* and *SVM* to predict flakiness with high precision and AUC; RQ1.2 confirms that this holds under stricter labeling; and RQ3 shows that this predictive power is not an artifact of opaque features, but stems from a small, interpretable set of tokens that act in two directions. The *flakiness markers* identified in Section 6.5, namely concurrency primitives (*async*, *await*), expectation-based synchronisation (*expectation*, *fulfill*), error-propagation constructs (*throws*, *try*), and explicit timing dependence (*timeout*, *wait*, *now*), map onto root causes of flakiness reported in the broader literature, including concurrency, time dependency, and resource management [7, 16]. The *stability markers* represent a complementary signal: the vocabulary of tests that assert plainly and synchronously, which the model uses as evidence against flakiness.

These findings relate to, but are not identical to, the vocabulary reported for other languages. Pinto et al. [20] found that the tokens most strongly associated with flakiness in Java were related to remote task execution and event queues (e.g., *job*, *table*, *id*, *action*), pointing to a vocabulary centered on distributed/asynchronous infrastructure. In our Swift dataset, the concurrency-related signal is expressed through language-level primitives (*async*, *await*, *async throws*) and asynchronous assertion patterns (*expectation*, *fulfill*, *throws*). A distinctive feature not reported for Java is the presence of stability markers: tokens whose *absence* in concurrency-heavy tests raises the flaky prediction. UI interaction tokens (*accessibilityidentifiers*), while present in the corpus, do not appear among the top-20 discriminative features in this dataset; this is consistent with the project composition, where 60% of the flaky tests (55 of 91) originate from non-UI projects (networking/server-side, persistence, and developer tooling), where concurrency vocabulary dominates. This suggests that while the underlying phenomenon (non-determinism caused by concurrency, timing, and external state) is consistent across ecosystems, its lexical manifestation is shaped by the language's idioms and the dominant testing style of the projects studied.

Two further experiments bound this. On the natural distribution (training balanced, test folds left at the real 0.4% rate) Recall and ranking hold up (0.82, AUC 0.947) but Precision collapses to 0.041: catching 15 flaky tests costs 374 false alarms. Leave-one-project-out gives a macro-averaged F1 of 0.71 and AUC of 0.90, against 0.86 and 0.95 when a project may appear on both sides of the split, ranging from 0.91 on *GRDB.swift* to 0.20 on *firefox-ios*, the most UI-heavy subject; part of the headline figure is thus within-project vocabulary. A binary CI gate would therefore be unusable, but the model's *ordering* is not: average precision is 0.31 against a 0.004 baseline, and half of the ten highest-scored tests are genuinely flaky. The realistic use is a prioritiser that spends a limited rerun budget where it pays off, which keeps the approach attractive for the expensive UI suites [25], with weaker guidance expected on projects unlike the training corpus.

8 Threats to Validity

Construct validity. Flaky labels are derived from two sources with complementary noise profiles: commit and pull-request mining depends on developers explicitly documenting non-determinism and may admit both false positives and false negatives; local re-execution used a single uncontrolled macOS machine, which may have masked latent flakiness or introduced spurious failures. To assess the impact of label noise, we replicated the full evaluation using only the 25 re-execution-confirmed flaky tests, for which the label is unambiguous. Four of five classifiers remained within two percentage points of their full-dataset F1 on this stricter subset, supporting the soundness of the hybrid labeling strategy.

Internal validity. The severe class imbalance required random undersampling, which also balances the test folds and so makes every Precision reported in Section 6 an optimistic reading of production behaviour; Section 7 quantifies the gap (Precision 0.041 at the natural 0.4% base rate) and the cross-project one. Repeating the undersample 30 times removes the dependence on any single draw, and stratified 5-fold cross-validation mitigates overfitting, but the small flaky-test pool inherently constrains generalization.

External validity. All 15 projects are public open-source Swift repositories on GitHub. The limited availability of large Swift projects with mature automated test suites, an ecosystem constraint rather than a sampling choice, restricts project diversity. Findings may not transfer to proprietary codebases or to Swift projects in domains or testing styles substantially different from those studied here.

Conclusion validity. To avoid misleading conclusions in the context of imbalanced data, we reported multiple evaluation metrics: precision, *recall*, *F1-Score*, Matthews correlation coefficient (*MCC*), and area under the ROC curve (*AUC*).

9 Conclusion

We presented the first empirical study of vocabulary-based static prediction of flaky tests in *Swift*, a language with a native *async*/*await* concurrency model and a strong reliance on UI testing. On 91 flaky and 22,349 non-flaky tests from 15 open-source projects, source-code tokens alone proved effective predictors of instability: *Random Forest* reached an F1-Score of 0.86 and AUC of 0.95, far above every trivial baseline (MCC = 0.75 vs. 0.08), and the result holds when the positive class is restricted to re-execution-confirmed labels. The discriminative tokens are grounded in Swift's concurrency semantics, with *async*, *await*, *throws*, *expectation* and *timeout* marking flakiness and *xctassertequal*, the vocabulary of plainly synchronous tests, marking its absence. The errors delimit the approach: flakiness hidden in shared infrastructure is missed and deterministic uses of async constructs are flagged, both distinctions being semantic rather than lexical. Since precision falls sharply at the natural base rate, the practical value lies in ranking rather than gating, and closing the remaining gap will require semantic and structural signals such as data-flow-aware features and interprocedural token propagation.

Artifact Availability

The replication package is available at https://doi.org/10.5281/zenodo.21881133. It contains the labeled corpus with per-test provenance, the scripts that regenerate every table reported here, and a Dockerfile that runs them with nothing installed locally. Our code and curated data are released under CC BY 4.0; the Swift test excerpts remain under the licenses of the projects they were taken from, as recorded in the package's `LICENSE`.

References

[1] Azeem Ahmad, Xin Sun, Muhammad Rashid Naeem, Yasir Javed, Mohammad Akour, and Kristian Sandahl. 2025. Understanding Flaky Tests Through Linguistic Diversity: A Cross-Language and Comparative Machine Learning Study. *IEEE Access* 13 (2025).

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *43rd International Conference on Software Engineering*.

[3] Keila Barbosa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda. 2022. Test flakiness across programming languages. *IEEE Transactions on Software Engineering* (2022).

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: automatically detecting flaky tests. In *40th International Conference on Software Engineering*.

[5] Alexander Berndt, Vekil Bekmyradov, Rainer Gemulla, Marcus Kessel, Thomas Bach, and Sebastian Baltes. 2026. Can We Classify Flaky Tests Using Only Test Code? An LLM-Based Empirical Study. *arXiv preprint arXiv:2602.05465* (2026).

[6] Sentry Cocoa Contributors. 2026. https://github.com/getsentry/sentry-cocoa.

[7] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[8] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. 2022. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE TSE* (2022).

[9] Martin Gruber and Gordon Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 82–92.

[10] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *37th International Conference on Software Engineering - Volume 2*.

[11] Mozilla iOS Contributors. 2025. Pull Request #27270. https://github.com/mozilla-mobile/firefox-ios/pull/27270/.

[12] Gregory M. Kapfhammer. 2004. Software testing. In *The Computer Science Handbook*. CRC Press.

[13] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *42nd International Conference on Software Engineering*.

[14] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*.

[15] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects *(Proceedings - International Symposium on Software Reliability Engineering, ISSRE)*. IEEE Computer Society.

[16] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[17] John Micco. 2017. The State of Continuous Integration Testing @Google.

[18] Mozilla. 2026. Firefox for iOS: Open-Source Web Browser for iOS. https://github.com/mozilla-mobile/firefox-ios

[19] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (2021).

[20] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *17th International Conference on Mining Software Repositories*.

[21] 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 *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[22] Shanto Rahman, Abdelrahman Baz, Sasa Misailovic, and August Shi. 2024. Quantizing large-language models for predicting flaky tests. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 93–104.

[23] Shanto Rahman, Saikat Dutta, and August Shi. 2025. Understanding and improving flaky test classification. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1345–1371.

[24] Rafael Rampim Soratto and Marco Aurélio Graciotto Silva. 2023. Vocabulary of Flaky Tests in Javascript. In *XXII Brazilian Symposium on Software Quality*.

[25] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1585–1597.

[26] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–311.

[27] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 534–538.

[28] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know You Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* (2021).