# Understanding and Improving Flaky Test Classification

SHANTO RAHMAN, The University of Texas at Austin, USA
SAIKAT DUTTA, Cornell University, USA
AUGUST SHI, The University of Texas at Austin, USA

Regression testing is an essential part of software development, but it suffers from the presence of *flaky tests* – tests that pass and fail non-deterministically when run on the same code. These unpredictable failures waste developers' time and often hide real bugs. Prior work showed that fine-tuned large language models (LLMs) can classify flaky tests into different categories with very high accuracy. However, we find that prior approaches over-estimated the accuracy of the models due to incorrect experimental design and unrealistic datasets – making the flaky test classification problem seem simpler than it is.

In this paper, we first show how prior flaky test classifiers over-estimate the prediction accuracy due to 1) flawed experiment design and 2) mis-representation of the real distribution of flaky (and non-flaky) tests in their datasets. After we fix the experimental design and construct a more realistic dataset (which we name **FlakeBench**), the prior state-of-the-art model shows a steep drop in F1-score, from 81.82% down to 56.62%. Motivated by these observations, we develop a new training strategy to fine-tune a flaky test classifier, **FlakyLens**, that improves the classification F1-score to 65.79% (9.17pp higher than the state-of-the-art). We also compare FlakyLens against recent pre-trained LLMs, such as CodeLlama and DeepSeekCoder, on the same classification task. Our results show that FlakyLens consistently outperforms these models, highlighting that general-purpose LLMs still fall short on this specialized task.

Using our improved flaky test classifier, we identify the important tokens in the test code that influence the models in making correct or incorrect predictions. By leveraging attribution scores computed per code token in each test, we investigate the tokens that have higher impact on the flaky test classifier's decision-making per flaky test category. To assess the influence of these important tokens, we introduce adversarial perturbation using these important tokens into the tests and observe whether the model's predictions change. Our findings show that, when introducing perturbations using the most important tokens, the classification accuracy can change by as much as -18.37pp. These results highlight that these models still struggle to generalize beyond their training data and rely on identifying category-specific tokens (instead of understanding their semantic context), calling for further research into more robust training methodologies.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: flaky tests, flaky test classification, large language models

**ACM Reference Format:**
Shanto Rahman, Saikat Dutta, and August Shi. 2025. Understanding and Improving Flaky Test Classification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 320 (October 2025), 27 pages. https://doi.org/10.1145/3763098

## 1 Introduction

Regression testing is an integral part of the software development process but is plagued by the presence of flaky tests. A *flaky test* is a test that can non-deterministically pass or fail when run on the same version of code [34]. When a flaky test fails after a developer makes changes, the developer cannot be sure if the failure is due to any fault introduced in the changes, or if it is a

Authors' Contact Information: Shanto Rahman, The University of Texas at Austin, Austin, USA, shanto.rahman@utexas.edu; Saikat Dutta, Cornell University, Ithaca, USA, saikatd@cornell.edu; August Shi, The University of Texas at Austin, Austin, USA, august@utexas.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART320
https://doi.org/10.1145/3763098

failure due to flakiness. A test can be flaky for many reasons, such as relying on asynchronous waits, concurrency, test-order dependencies, and more [19, 34]. Flaky tests are wide-spread, in both open-source software and in industry code [19, 22, 27, 34–36, 51].

A common approach to detect flaky tests is to simply rerun tests many times until observing both pass and fail outcomes [1]. Prior research proposed many different ways of proactively detecting flaky tests by rerunning tests in different ways such as by adding delays during execution [31, 41], changing test order [29, 47, 49], adding stress to the environment using resource-intensive tasks [45], and more [16–18, 44]. However, re-running tests remains a cost-intensive strategy, making it hard to scale to large test suites.

**Prior ML-based approaches.** To reduce the cost of flaky test detection, prior work proposed the use of machine learning (ML) to predict which tests are flaky without needing to re-run them. Alshammari et al. trained traditional machine learning classifiers to predict flakiness using code and runtime features from a large dataset of flaky tests [14], while Pinto et al. extracted common code features from test code to use for predicting flaky tests [39]. Recent work investigated the usage of large language models (LLMs) to similarly predict flaky tests based on just test code. Fatima et al. proposed Flakify, a classifier for predicting whether a test is flaky by just parsing the test code, which they created by fine-tuning the pre-trained CodeBERT model [4] using a large dataset of flaky tests [20]. Akli et al. proposed FlakyCat, a classifier that can determine the flaky test category for a given flaky test, e.g., that the test is flaky due to reliance on asynchronous waits, by leveraging an LLM and few-shot learning [13]. Rahman et al. later improved upon both these prior models by fine-tuning an LLM to predict whether a test is flaky and also predict the category of flakiness, but with lower cost due to quantization [40]. All these prior approaches reported that LLM-based flaky test classifiers are highly accurate.

Despite the success of these LLM-based classifiers at their respective tasks, it is unclear as to *why* exactly these classifiers can have such high accuracy. These classifiers make their predictions solely based off of parsing test code, so it is surprising that they can perform so well using such limited information. Intuitively, after training, the classifiers must be picking up on specific patterns or tokens within the test code to help them make their decisions as to whether the test is flaky, or to determine the flaky test category. For example, flaky tests that belong to the asynchronous wait (Async Wait) category (meaning their execution waits a fixed time for some other service to finish running but does not synchronize with that other service) likely make calls to methods like `wait()` or `sleep()` to set the waiting time. We hypothesize that such classifiers primarily rely on syntactic clues (e.g., token frequency linked to specific categories) rather than achieving a deeper, more robust understanding of flakiness.

## 1.1 Our Work

**Improved flaky test classifier.** We first re-evaluate the experiment designs of prior approaches that perform LLM-based flaky test prediction. We find two key shortcomings. First, prior research used a smaller dataset of flaky tests containing *only* flaky tests or a small number of non-flaky tests [20]. However, such datasets are not representative of the real-world distribution of flaky tests, where a majority of tests are typically non-flaky. Second, we find an experimental design flaw in prior work that led to an over-estimation of the classifier's F1-score. We informed the authors about the data leakage, and they acknowledged our findings and updated their GitHub repository accordingly [8].

To mitigate these problems, we propose a new dataset, **FlakeBench**, that contains 280 flaky and 8294 non-flaky tests from real-world projects for training flaky test classifiers. We also propose a new flaky test classifier, **FlakyLens**, using a new training strategy to apply on a pre-trained

LLM for training a flaky test classifier. This new strategy improves the performance of flaky-test classification by leveraging our insights into the distribution and characteristics of the domain of flaky tests, namely that there are very few flaky tests relative to non-flaky tests.

Our results show that FlakyLens outperforms existing flaky test classification approaches, including the corrected version of Flakify and FlakyCat, achieving notable improvements in F1-score (i.e., 9.17pp and 13.79pp, respectively).

**Comparing against pre-trained LLMs.** We also compare FlakyLens against the latest open-source pre-trained LLMs by prompting them to classify flaky tests. Our goal is to not just evaluate how well a pre-trained LLM is at performing zero-shot classification tasks, but also to see whether pre-trained LLMs rely on similar important tokens as FlakyLens in making their decisions. We similarly rely on attribution scores computed per token to extract these important tokens per test from these pre-trained LLMs.

FlakyLens achieves significantly higher accuracy in flaky test prediction, correctly classifying 8468 out of 8574 tests, with the macro-average F1-score of 65.79%, whereas the best-performing zero-shot LLM reaches only 14.86%. In other words, FlakyLens achieves a 50.93pp improvement over the strongest pre-trained LLM baseline in terms of F1-score.

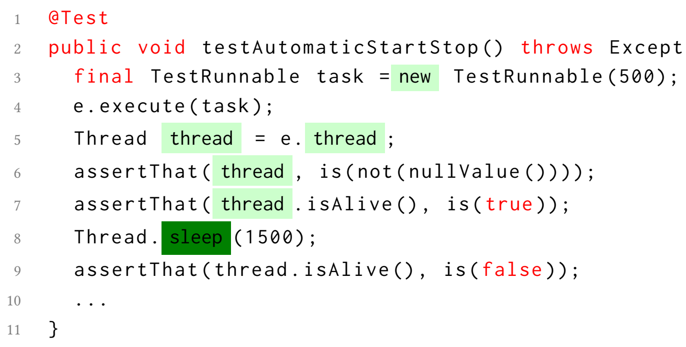
**Understanding flaky test classifiers.** While we find FlakyLens to be an improvement over existing flaky test classifiers, we would like to better understand how it makes its decisions based solely off of test code. We propose a means to extract the relevant tokens from test code that a LLM-based classifier uses for predicting flakiness, as well as to measure the impact of those tokens on the classifier's performance. We identify the important tokens that the model uses, namely the parsed test code tokens, using Integrated Gradients (IG) [43] to compute feature attribution scores. Tokens with higher attribution scores are likely the ones that the model puts higher emphasis on for its predictions. To further demonstrate the impact of these tokens with higher attribution scores, we leverage code perturbations based on prior work [21] that can introduce these tokens into test code while preserving semantics. If we find that FlakyLens can be "tricked" and mispredict tests that have been perturbed using relevant tokens, then those tokens have impact on how FlakyLens makes decisions.

Our results indicate that perturbation using important features can reduce the F1-score by up to 18.37pp. In contrast, altering the least important features results in a maximum impact of only 8.70pp. These results highlight how an LLM-based flaky test classifier like FlakyLens relies heavily on syntactic features (important tokens) instead of understanding the semantics of code (e.g., the context around such important tokens).

## 1.2 Contributions

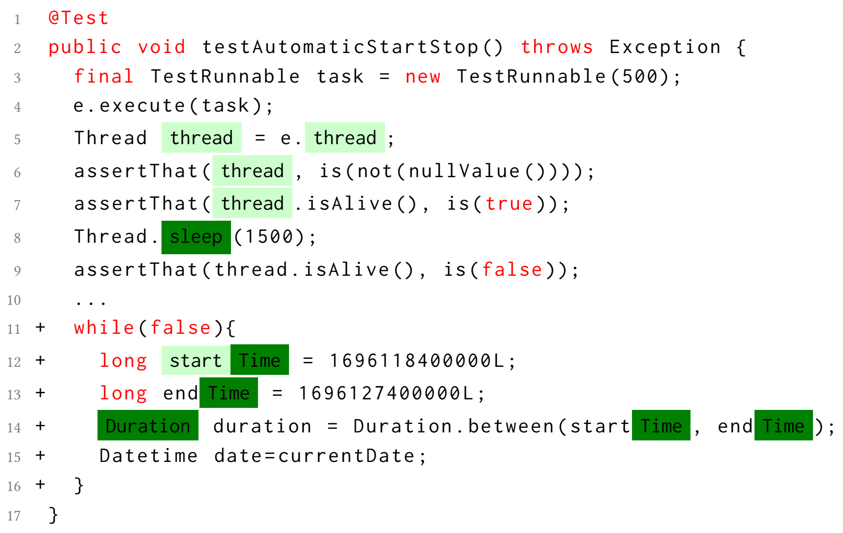
Our paper makes the following contributions:

- **FlakeBench Dataset:** We develop FlakeBench, a curated dataset of flaky and non-flaky tests that reflects real-world distributions, including the natural imbalance with more non-flaky tests, enabling more realistic classifier evaluation.
- **Identifying and Addressing Design Flaws:** We uncover critical experimental design flaws in existing flaky test classifiers, such as Flakify, which has been acknowledged by the authors of this past work after we reported our findings to them. Building on these insights, we propose FlakyLens, a strategy for fine-tuning CodeBERT to construct a more robust and accurate flaky test classifier.
- **Benchmarking Latest LLMs:** We evaluate state-of-the-art open-source generative LLMs on the flaky test classification task. We further investigate how these models make predictions using token-level attribution scores, offering insights into their decision-making behavior.



```
1    @Test
2    public void testAutomaticStartStop() throws Exception {
3      final TestRunnable task = new TestRunnable(500);
4      e.execute(task);
5      Thread thread = e.thread;
6      assertThat(thread, is(not(nullValue())));
7      assertThat(thread.isAlive(), is(true));
8      Thread.sleep(1500);
9      assertThat(thread.isAlive(), is(false));
10     ...
11   }
```

(a) Original: Async Wait flaky test



```
1    @Test
2    public void testAutomaticStartStop() throws Exception {
3      final TestRunnable task = new TestRunnable(500);
4      e.execute(task);
5      Thread thread = e.thread;
6      assertThat(thread, is(not(nullValue())));
7      assertThat(thread.isAlive(), is(true));
8      Thread.sleep(1500);
9      assertThat(thread.isAlive(), is(false));
10     ...
11 +   while(false){
12 +     long startTime = 1696118400000L;
13 +     long endTime = 1696127400000L;
14 +     Duration duration = Duration.between(startTime, endTime);
15 +     Datetime date=currentDate;
16 +   }
17   }
```

(b) Predicted: Time flaky test (deadcode perturbation at Lines 11-16)

Fig. 1. Example of deadcode perturbation. Highlighted tokens indicate higher token importance, with deeper green shades representing greater token importance.

- **Interpretability and Robustness:** We develop a technique to better understand how FlakyLens makes its predictions by extracting important tokens from test code using attribution scores. We demonstrate their significance by applying semantics-preserving code perturbations and measuring their impact on classification accuracy.

## 2 Example

Figure 1a illustrates a simplified version of an asynchronous wait (Async Wait) flaky test from project netty [3], which is known to be flaky from prior work [13]. The `testAutomaticStartStop` is a unit test that verifies the automatic start and stop behavior of a task execution system. The test creates a `TestRunnable` object with a 500-millisecond duration on Line 3. The test then submits the object for execution on Line 4. Then, the test retrieves the thread used on Line 5 and asserts that this thread is not null and is actively running, confirming that a thread is correctly created for the task. The test then pauses for 1500 milliseconds on Line 8 to allow the task to complete. Finally, it verifies that the thread is no longer alive, confirming that the task has finished execution and the thread has terminated as expected.

This test is inherently flaky due to its dependence on timing and thread scheduling. In one execution, the task may finish within the expected time, causing the test to pass. In another execution, the thread may not complete in time, leading to a test failure. This flakiness stems from the use of threads and sleep-based waits. Intuitively, the presence of tokens such as `Thread` and `sleep` are critical for predicting the flakiness category.

Our goal is to determine which are the important tokens that a flaky test classifier uses when classifying the test. By analyzing token attribution scores, we find that our proposed flaky test classifier, FlakyLens, successfully highlights the expected tokens as influential in its predictions. When we analyze the tokens' attribution scores [43] for the original code in Figure 1a, we see that `sleep` (3.03), and `thread` (1.70) have the highest scores compared to other tokens.

To assess the impact of these important tokens, we apply semantics-preserving perturbation to the test code in Figure 1b. For instance, we take important tokens from a different flaky category, e.g., flaky tests of the Time category often include important tokens like `Time` and `Duration`. We inject these important tokens of the other category into the original test without changing its behavior at Lines 11-16. Specifically, we inject deadcode that should never be executed, but this deadcode contains these other important tokens. Our findings show that after applying this perturbation, we find `Time` and `Duration` to now have the highest attribution scores (5.12 and 3.49, respectively), followed by `sleep`, `thread`, also having mentionworthy scores (2.93, and 1.48, respectively). As a result, this test is predicted to be of the Time category. This change in prediction demonstrates that the identified important tokens play a meaningful role in the model's decision-making process for flaky test classification.

## 3 FlakeBench

We construct FlakeBench, a dataset designed to reflect the distribution of flaky and non-flaky tests observed in real-world projects. We begin by collecting a dataset of previously labeled flaky test categories, developed by Akli et al. in FlakyCat [13], where the categories are based on root causes defined by prior studies [19, 34]: Async Wait, Concurrency, Time, Unordered Collections, Test Order Dependency, Network, Randomness, and Resource Leak. We only consider flaky test categories that have at least 30 tests, similar to prior work [40], eliminating the Network, Randomness, and Resource Leak categories. However, this dataset does not contain non-flaky tests. As a result, this dataset is not representative of the real-world distribution of flaky tests, where a majority of tests are typically non-flaky.

### 3.1 Collecting Flaky and Non-Flaky Tests

To construct a more realistic dataset with representative non-flaky tests, we begin by taking all projects associated with the known Java flaky tests evaluated by FlakyCat [13]. We keep only the projects with at least 30 labeled flaky tests, resulting in an initial pool of 142 distinct projects. We eliminate some projects due to issues such as missing owners or invalid commit hashes, and finally retrieve and clone 97 of these projects. For each project, we check out the exact commit associated with the flaky test and collect all test files located in standard test directories (e.g., `src/test/java/`). Each test file represents a test class, which contains individual tests. We identify and extract the tests as methods annotated using the standard JUnit or TestNG testing framework test annotations (e.g., `@Test`, `@ParameterizedTest`, `@BeforeEach`), identifying a total of 179,403 candidate tests.

Let $\mathcal{T}_{\text{all}}$ be the set of all extracted tests. To obtain our set of non-flaky tests, we start by removing all known flaky tests. Then, for each other test, if they are in the same test class as a known flaky

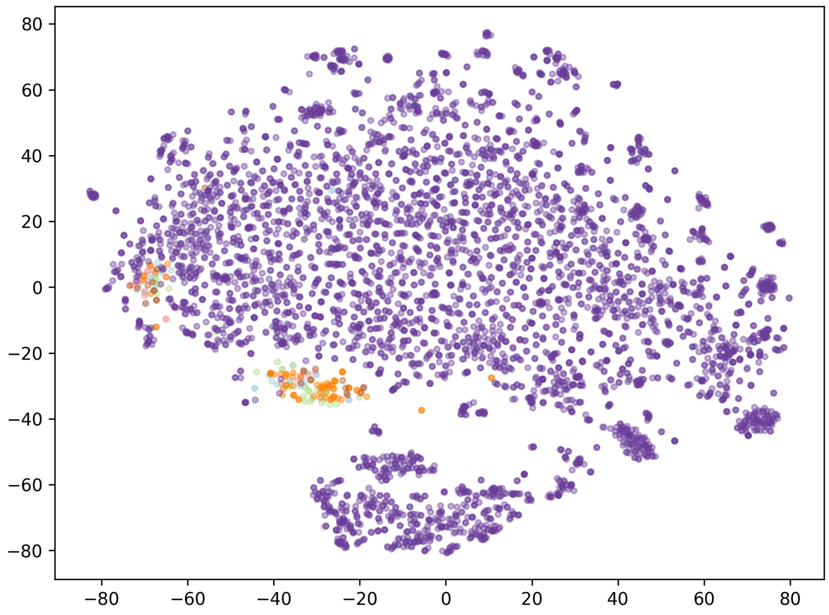


Fig. 2. t-SNE visualization of the data, where each point represents a test and colors indicate different categories. Each test's code embeddings are projected into a 2D space for visualizing clusters based on similarities in features.

test, we also remove those tests. This filtering ensures that overlapping or previously identified flaky contexts are excluded from the non-flaky pool. The remaining tests are considered as potentially non-flaky, $\mathcal{T}_{\text{nonflaky}}$, and we run those tests 100 times for additional confirmation that they are not flaky. For each selected test $t \in \mathcal{T}_{\text{nonflaky}} \subseteq \mathcal{T}_{\text{all}}$, we extract and store metadata including: project name, test class and test names, test method body with the method signature, and assigned label $\texttt{category}(t) = \texttt{non-flaky}$. This process yields a high-quality pool of realistic non-flaky test cases for training and evaluation purposes. Ultimately, FlakeBench contains a total of 8574 labeled tests, of which 280 are flaky (spread across five flaky test categories) and 8294 are non-flaky.

## 3.2 Distribution of Flaky and Non-Flaky Test Categories

Table 1 (column '#Tests') presents the breakdown of the number of tests per category in the dataset. In general, we see that each flaky test category contains significantly fewer instances than the non-flaky category. To further examine this imbalance, we visualize the dataset using a 2D t-SNE plot in Figure 2. In the plot, we convert each test into a 768-dimensional embedding (as produced by the CodeBERT architecture), which we project into 2 dimensions by taking the 2 most major features, for visualization purposes. Each point represents a test, which we color based on the flaky test category. The plot illustrates that flaky test categories tend to form overlapping clusters, indicating shared features, which makes the classification task more difficult.

We intentionally include a larger proportion of non-flaky tests to better reflect real-world settings, where flaky tests are relatively in smaller proportions. In prior work, Google reported that 1.5% of their tests are flaky [37], and Microsoft reported 4.6% of their tests are flaky [28]. In FlakeBench, we have 3.4% of tests labeled as flaky, as shown in Table 1, aligning with these previously reported distributions reported from company codebases. However, prior research, such as Flakify [20], uses

Table 1. Distribution of test categories across projects and test cases in FlakeBench. The dataset includes six categories of tests, including five types of flaky tests and one non-flaky category, totaling 8574 test cases across varying numbers of projects.

| ID | Category | #Projects | #Tests |
|---|---|---|---|
| 1 | Async. | 39 | 76 |
| 2 | Conc. | 17 | 37 |
| 3 | Time | 24 | 33 |
| 4 | UC | 28 | 41 |
| 5 | OD | 17 | 93 |
| 6 | Non-flaky | 97 | 8294 |
| Σ | - | - | **8574** |

a dataset where flaky tests are six times more frequent than non-flaky tests. Hence, their dataset is not as representative of real-world distributions and may unintentially bias trained models to predict most tests as flaky.

## 4 Improving Flaky Test Classifiers

We propose FlakyLens, a new LLM-based flaky test classifier. We first discuss the design flaws in previous work on LLM-based flaky test prediction, Flakify [20], which lead to data leakage issues. We then discuss how we fine-tune a pre-trained LLM to predict flaky test categories, forming FlakyLens.

### 4.1 Data Leakage Issue in Prior Work

Flakify is a prior LLM-based technique that predicts whether a test is flaky or not by analyzing just the test code, i.e., it classifies a test into either flaky or non-flaky categories [20]. Flakify is built by fine-tuning a pre-trained LLM, CodeBERT, on a dataset of tests labeled as flaky or non-flaky. The evaluation for Flakify follows a stratified k-fold cross-validation methodology, where they split the dataset across multiple folds with different training/validation/test sets. Each fold should train a new model using that fold's data (where in each fold the model's weights get updated across several epochs via back-propagation), and the final evaluation results report the average performance of the models across all folds.

However, we discovered a data leakage issue due to how the evaluation uses stratified k-fold cross-validation with shuffling enabled. In this setup, a test that appears in the training set of one fold may later appear in the test set of another fold. While this setup is common practice in machine learning evaluation, the critical flaw lies in how the pre-trained CodeBERT model (`auto_model`) is loaded across folds. Ideally, for each fold $i$, a fresh instance of the pre-trained model $M^0$ should be loaded and fine-tuned on $D^i_{\text{train}}$ ($D$ denotes dataset), producing a new model $M^i$. Unfortunately, the previous evaluation initialized the model only once and reused it across all folds without reloading:

$$M^1 \xrightarrow{\text{fine-tune}} M^2 \xrightarrow{\text{fine-tune}} \cdots \xrightarrow{\text{fine-tune}} M^K.$$

This configuration introduces a significant issue because PyTorch-based models retain learned knowledge in memory, meaning that gradient updates from previous folds persist in subsequent iterations [38]. As a result, knowledge from previous folds carries over into subsequent folds, effectively leaking information into the evaluation process. This unintended leakage leads to evaluation results that indicate Flakify achieving higher and higher F1-scores across consecutive folds, ultimately reaching an extremely high F1-score by the end.

Once we corrected this issue by ensuring the model is reloaded independently for each fold, we observed that the actual prediction performance was significantly lower than previously reported. We informed the authors about the data leakage, and they acknowledged our findings and updated their GitHub repository accordingly [8].

## 4.2 FlakyLens

We introduce FlakyLens, a new fine-tuned model designed to mitigate data leakage issues while also addressing the challenges that come with a more realistic dataset.

*4.2.1 Project-Wise Disjoint Training/Test Split.* To ensure fair evaluation and prevent data leakage, we partition the dataset into a collection of project-wise disjoint training/test sets, ensuring that tests from the same project never appear in both training and test sets. This partitioning simulates a more realistic model usage scenario where a developer applies a flaky test classifier to a new project, and no tests from that project was ever used when training that flaky test classifier. This project-based partitioning helps the model learn without relying on prior knowledge from the same project, thereby enabling better generalization to unseen projects. While partitioning into the training/test sets, we enforce that each test set contains at least five tests from every flaky test category.

*4.2.2 Fine-Tuning Process.* For each fold $i$, we fine-tune a pre-trained CodeBERT model $M^0$ on $D^i_{\text{train}}$, producing a fold-specific model $M^i$. We ensure that $M^0$ is reloaded afresh before each fold's fine-tuning process to eliminate residual learning and prevent data leakage across folds:

$$M^0 \xrightarrow{\text{train on } D^i_{\text{train}}} M^i.$$

To construct a flaky test classifier that predicts the flaky test category for a test, we introduce a linear classification head on top of the model's final encoder output, producing logits for the six target flaky test categories. Given the heavy class imbalance in $D^i_{\text{train}}$, particularly the dominance of non-flaky tests, we compute class-balanced weights using scikit-learn's `class_weight='balanced'` heuristic. This approach assigns higher weights to under-represented classes based on the inverse of their frequency, normalized across all categories. We use these weights during training to mitigate bias toward majority classes.

To further address the imbalance problem, we replace the standard cross-entropy loss with Focal Loss [32], which emphasizes harder-to-classify examples. The loss is defined as:

$$\text{FL}(p_t) = -\alpha_t(1-p_t)^\gamma \log(p_t)$$

where $p_t$ is the predicted probability of the true class, $\gamma = 2$ controls the focus on misclassified instances, and $\alpha_t$ is the class-specific weight derived from the balanced weighting scheme.

To mitigate overfitting and promote generalization during fine-tuning, we incorporate several regularization techniques. First, we apply L2 weight decay (with weight decay coefficient $\lambda = 0.01$) in the Adam optimizer [50]. L2 regularization penalizes large weights, thereby encouraging the model to learn simpler and more generalizable representations rather than memorizing the training data. Second, we use a dropout rate of 0.3, which randomly deactivates a fraction of neurons during training to prevent co-adaptation and improve generalization.

We train for a maximum of 30 epochs, which provides sufficient opportunity for the model to converge while avoiding unnecessarily long training runs that could lead to overfitting. To further mitigate this overfitting, we adopt early stopping with a patience of 5 epochs, monitoring the validation loss to stop training when we no longer observe any improvement. We set the learning rate used between epochs to $1 \times 10^{-5}$; we found this rate to have consistently performed well during tuning compared to other values that we tried.

## 5 Understanding Flaky-Test Classifiers

We aim to determine the impact of the important features that an LLM-based flaky test classifier uses to make classification decisions. Since the LLM-based flaky test classifier uses tokens from test code as input features for their prediction [13, 20, 40], we need to analyze which tokens are the most important. We analyze which tokens contribute the most towards the final prediction for each flaky test category (including the "non-flaky" category). Our intuition is that, if the presence of certain tokens is particularly relevant for the flaky test classifier to decide on a category, then we can apply some "adversarial perturbation" to existing tests, adding code built around those relevant tokens that still preserves the same semantics of the test before, as a means to "attack" the flaky test classifier, forcing it to mispredict towards a specific category. If we can observe substantial mispredictions based on our "attack", we can more confidently determine what a flaky test classifier is actually using as information to make its decisions.

### 5.1 Token Importance

First, to identify some important tokens, we calculate the token attribution scores using Integrated Gradients (IG) [43]. IG is one of the interpretability techniques implemented in Captum [26]. Captum is an open-source library for model interpretability focused on PyTorch [24]. We select IG for its context-sensitive, gradient-based explanations that align closely with the model's decision. IG measures token influence by calculating gradients from a baseline to the actual input, providing a continuous, directional importance measure [43]. Other explainability techniques, such as LIME, cannot always capture the exact contribution of features, especially in complex or nonlinear models, because they rely on locally linear approximations around each input data point [42]. Another popular technique, SHAP, often uses sampling-based estimations or approximations when exact Shapley values are computationally infeasible, particularly in deep models [33]. This reliance on approximations can introduce minor inconsistencies and may violate implementation invariance under certain conditions [33].

*5.1.1 Computing Token Attribution Score.* We start by extracting tokens from the given test set data, by using the CodeBERT tokenizer [4], resulting in 512 tokens for each test. If the test code has fewer than 512 tokens, we pad the remaining with empty tokens at the end. If the test code has more than 512 tokens, we truncate to keep only the first 512 tokens. We apply this padding and truncating strategy consistently in both fine-tuning and evaluation, which is the same as done in prior work [20]. We observe that most tests in our dataset contain at most 512 tokens (86.4% in training set, 88.4% in validation set, and 74.7% in test set). Then for these tokens, we extract the embeddings (token vectors) from the fine-tuned model and use these embeddings to calculate IG. To calculate IG, we consider a zero vector baseline as a neutral reference point. We then create interpolated states between the baseline and the input, allowing gradients to be computed across these steps. Finally, we aggregate the gradients for each token to measure each token's contribution from baseline to input [46]. These gradients together represent the attribution score per token, a quantitative measurement that indicate the degree to which this token influences the model's prediction. We are most interested in the tokens that have the highest attribution scores, indicating tokens that are the most important tokens.

*5.1.2 Token Post-Processing and Detokenization.* While following the previous steps obtain attribution scores per token, many tokens require de-tokenization and additional post-processing. This step is essential, because the goal is to identify meaningful keywords within the tests that influence the model's predictions. After tokenization, tokens often lose their original form, e.g., "Equals" being split into "Equ" and "als", making it difficult to interpret the intended word. This lack of meaning complicates any understanding of the influential tokens, and when we later introduce perturbation using such tokens, the injected tokens may not appear realistic.

We observe that the CodeBERT tokenizer marks new words starting after whitespace with the symbol $\dot{G}$. We look for $\dot{G}$ and merge tokens that do not start with a special character or capital letter. For example, we can merge "Equ" and "als" to form "Equals", because the second token, "als" starts with a lowercase character.

More precisely, for each consecutive tokens $(t_i, t_{i+1})$, we define the following conditions for merging the tokens:

- New Word Indicator: If a token $t_i$ begins with a special character $\dot{G}$ (indicating a new word), we merge $t_{i+1}$ if that $t_{i+1}$ is not a punctuation or a symbol.
- Lowercase Merge Condition: If $t_{i+1}$ begins with a lowercase, and $t_i$ does not contain any punctuation or special characters, the tokens are merged.

After obtaining the merged tokens, we post-process the tokens to remove stop words and punctuation. This step is necessary because tokenization often results in extraneous characters like dots (.), square brackets ([]), semicolons (;), etc. We remove all tokens that consist of just common stop words [20] and punctuation marks. Then we compute the attribution score for each merged token by summing the attribution scores of the individual tokens that make up the merged token.

*5.1.3 Computing Model Confidence Score.* We compute the confidence score of the fine-tuned model for each test. A confidence score reflects how certain the model is in classifying a test as belonging to a specific category. To obtain this score, let $M$ be a fine-tuned model trained to classify test methods into multiple categories, denoted as $C_j \in \{C_1, C_2,...,C_n\}$. The model, $M$, processes a token sequence $T = (t_1, t_2, \ldots, t_n)$ for each test, with an attention mask $A = (a_1, a_2, \ldots, a_n)$. We compute the confidence score using the following two steps:

(1) **Logit Calculation and Probability Distribution.** We compute logit vector $L = (l_1, l_2, \ldots, l_n)$, where each $l_j$ represents the score for class $C_j$, formally defined as:

$$L = M(T, A)$$

We then transform the logits $L$ into a probability distribution $p_j = (p_1, p_2, \ldots, p_k)$, where each $p_j \in [0, 1]$ represents the probability of class $C_j$, and the sum of all probabilities is 1.

(2) **Model Confidence Score and Predicted Class.** We define the confidence score $C_{\text{T}} \in [0, 1]$ as the maximum probability among all classes for a test:

$$C_{\text{T}} = \max_{j \in \{1,\ldots,k\}} p_j$$

This value represents the model's confidence in its most probable classification. Finally, the classification of the input is given by selecting the class that has the maximum probability score.

*5.1.4 Weighted Attribution Score.* Using a merged token's attribution score and the model's confidence score, we compute a weighted attribution score. The reason we perform this step is because attribution scores are aggregated for each token when it appears across multiple tests. If a test's prediction is accurate but has a lower confidence score, then the token should receive less priority than if it came from a test with a higher confidence score in correctly predicting the flaky test category. Therefore, we multiply the attribution score of a merged token by the model confidence score to adjust their impact accordingly.

Finally, we multiply this score for each token by the logarithm of frequency with which that token appears across multiple correctly classified test cases. This approach ensures that tokens that are both highly attributed and consistently present are assigned higher importance.

## 5.2 Data Perturbations

With the most important tokens based on weighted attribution scores, we design five different perturbation techniques, namely, deadcode, print statement, variable renaming, multi-line comments, and single-line comment, inspired by the prior research [21] and our own code analysis. The objective is to see whether additional code that utilizes important tokens yet does not affect test semantics can still affect the model's prediction. To construct the perturbed code to inject, we take the important tokens we want to inject and prompt an LLM to generate the desired type of perturbation using these tokens. Figure 3 shows a template of the base input and Figure 4 shows the system prompt we use to generate the perturbations. We provide more details concerning our prompts on our website [12].

```
Modify the provided Java test code to incorporate Deadcode, Print
Statement, Variable Rename, Single-Line Comment, and Multi-Line
Comment elements, using the specified token list.

Test Code: {test_code}

Token List: {token_list}
```

Fig. 3. Base prompt for generating perturbation.

```
As an expert in Java code modification, your task is to inject
adversarial perturbation into the given test code. There are five
types of adversarial perturbation to be added: Deadcode, Print
Statement, Variable Rename, Single-Line Comment, and Multi-Line
comment. Below are instructions for each type, along with an example:

Deadcode: Add a segment using while(false) { ... }. Insert code within the loop using
     the provided tokens.
Print Statement: Use System.out.println(...) to add a print statement. Populate the
    statement using the given token list.
Variable Rename: Rename the top three most frequently used variables based on the
    token list.
Single-Line Comment: Add a single line of Java code using the token list and comment
    it out with //.
Multi-Line Comment: Add a multi-line comment that does not necessarily need to follow
     Java syntax. Use /* ... */ to create this comment and
include content from the token list.
```

Fig. 4. System prompt for generating perturbation.

```
1    @Test
2    public void testBacklogLimiter() {
3       long duration = runWithRate(2 * RateLimiting.DEFAULT_MAX_PARALLELISM,-1.0,
4           new DelayFn<Integer>());
5       Assert.assertThat(duration,greaterThan(2 * DelayFn.DELAY_MS));
6  +    System.out.println("checking␣the␣wait␣time␣using␣Thread.sleep");
7    }
```

Fig. 5. Example of print statement perturbation (Line 6). Original: Time, Predicted: Async Wait

```
1      @Test
2      public void testSerialize() {
3        request = new GetUserIdSerializableRequest();
4        request.setOperation(GETUSERID);
5        request.setInfoField1("nobody@amazon.com");
6        request.setInfoField2("AMZN");
7        String requestString = serializer.encode(request);
8        assertEquals("{\"infoField2\":\"AMZN\"}", requestString);
9    +   /** {
10   +   * Timer repetition. Date : date.now() Time: 10:01:05.
11   +   * dayOfweek=saturday.
12   +   * }
13   +   **/
14     }
```

Fig. 6. Example of multi-line comments perturbation (Lines 9-13. Original: Unordered Collections, Predicted: Time

```
1    @Test
2    public synchronized void testLockExpiration() {
3      ...
4      lock = lockMgr.lock(...);
5      long remaining = lock.getSecondsRemaining();
6      if (remaining <= hint) {
7        try {
8           wait(remaining * 2000);
9        } catch (InterruptedException ignore) {
10       }
11       ...
12       assertFalse(message, lock.isLive());
13     } else {
14         throw new NotExecutableException("timeout␣hint.");
15     }
16 +   //System.out.println("Date Timestamp" + date.now());
17   }
```

Fig. 7. Example of single-line comment perturbation (Line 16). Original: Async Wait, Predicted: Time

*5.2.1 Deadcode.* Deadcode perturbation involves injecting “deadcode” that includes the important tokens related to another flaky test category into the test code, where this deadcode is never actually executed when the test runs, such as the code being contained within an `if` statement with a condition that is never satisfied. The goal is to observe whether the model can effectively identify the non-executing code segments, where the proper behavior is to ignore the deadcode. We inject the generated code into the original test code at the end. Figure 1 shows an example of deadcode perturbation, injected as Lines 11-16.

*5.2.2 Print Statement.* Print statement perturbation involves injecting a print statement (`System-.out.println` method call in Java) that tries to print out the important tokens of other flaky test categories into the test code. Print statements generally have no impact on the program’s logic. However, the model that does not properly consider the impact of print statements on semantics may then be misled by the tokens used within the statement. We inject this print statement at the end of the test. In Figure 5 on line 6, we illustrate injecting a print statement using important tokens for the Async Wait flaky test category, even though the test is actually of the Time flaky test category.

*5.2.3 Variable Renaming.* Variable renaming perturbation involves refactoring variable names to new names while keeping the program semantically the same without occuring any syntax error. We rename the three most frequently occurring variables in the test to instead be one of the important tokens of another flaky test category. We only rename the most frequently appearing variable, using only the important token with the highest weighted attribution score of the chosen other flaky test category. We do not go to the extreme of renaming all variables in the test code.

*5.2.4 Multi-Line Comments.* Multi-line comments perturbation involves adding comments that span multiple lines using important tokens of another flaky test category. Since comments are meant for human readers and not for execution by machines, adding these comments do not change test semantics, and the model should ignore any text within the multi-line comments. Figure 6 shows an example of multi-line comments (Lines 9- 13).

*5.2.5 Single-Line Comment.* Single-line comment perturbation involves adding a single line of comment. Similar to comments spanning multiple lines, these comments do not change test code semantics, and the model should ignore them. Figure 7 on Line 16 shows an example of a single-line comment that contains other important tokens.

Each of these perturbation techniques help in evaluating different semantic aspects of code analysis tools, from their ability to ignore irrelevant data to their capacity to focus on functionally critical components of the code.

## 6 Experimental Setup

### 6.1 Research Questions

We address the following research questions:

- **RQ1**: How does FlakyLens perform compared to existing flaky test classifiers?
- **RQ2**: How do the latest generative LLMs compare to FlakyLens at classifying flaky tests?
- **RQ3**: What are the important tokens that influence the model’s prediction?
- **RQ4**: How critical are the identified important tokens to the model’s predictions when they are injected into test code without changing semantics?

We address RQ1 to evaluate the effectiveness of FlakyLens by comparing with other existing fine-tuned flaky test classifiers. We address RQ2 to evaluate the zero-shot performance of the latest generative LLMs at this specific task of predicting flaky test categories. We also analyze the agreement and disagreement among different models. Specifically, we want to see whether they highlight the same tokens as important and how consistently do they agree across different tests and flaky test categories. To understand the key factors influencing model predictions, we address RQ3 that investigates the important tokens that have higher weighted attribution scores, indicating their influence on the model's decision-making. Finally, RQ4 shows the impact of these tokens on actual prediction. The intuition behind this analysis is to determine whether models rely on certain tokens even when they do not alter the test's semantics, as well as to quantify the extent of this influence.

## 6.2 Evaluation Setup

We design our setup so that tests from the same project do not appear in both the training and test sets of the same fold. For our evaluation, we choose to conduct a four-fold cross-validation methodology. For each fold, we first take the list of unique projects and randomly select a fixed number of non-overlapping projects as the test set, and the remaining projects for the training set. To ensure that each test set is representative enough for evaluation purposes, we repeatedly resample projects for the test set until we obtain one that includes at least four tests per category (four is the largest number of tests we can get for the category with the fewest number of tests, such that we maintain a balanced number of tests of that category across all folds). We also ensure that the same test does not appear in both a training set and test set per fold, as well as not appearing in multiple test sets across all folds.

For each fold, we train a classifier using that fold's training set and evaluate it on the corresponding validation set (the training set of each fold gets further divided into training and validation set when fine-tuning the model) across a maximum of 30 epochs (Section 4.2.2) to obtain the best model for that fold. We measure the effectiveness of the model by evaluating on the test set for that fold. We use the same data split across all subsequent experiments, including experiments with all recent LLMs, to ensure a fair comparison. For each fold, we maintain the training/validation/test split ratios as 50%, 20% and 30%, respectively. The distribution of tests across the flaky test categories per fold is preserved across the sets. For example, the percentage of tests in the Async Wait category across training, validation, and test sets per fold are 0.67%, 0.65%, and 1.44%, respsectively.

## 6.3 Open-Source Generative LLMs

We use six other LLMs for comparison purposes: two *Meta* models, namely CodeLlama-13B and Llama-3-8B; two *Google* models, namely Codegemma-7B and Gemma-7B; one Qwen model, namely Qwen2-7B; and one *deepseek-ai* model, namely Deepseek-Coder-V2-Lite (16B). We use the instruct version of these models from Huggingface, because instruct models are fine-tuned to handle user instructions. We do not fine-tune these models ourselves for the specific flaky test classification task, as we want to evaluate how well these more general-purpose LLMs perform in a zero-shot setting. It would be interesting future work to fine-tune these models as well.

For each model, we consider the corresponding model's tokenizer to generate tokens from the given data. We generate a template that contains a base prompt with a system prompt (for providing more context). We provide an example of a base prompt and system prompt for this step on our website [12]. We then use the Huggingface API [5] to prompt the LLMs for predicting the flaky test category of a given test.

To compute token-level attribution scores with these pre-trained models, we leverage Integrated Gradients (IG) to identify which input tokens most influence the predictions of each model. We begin by tokenizing the input prompt and test code using the appropriate tokenizer for the model in use, ensuring that all required inputs, such as the attention mask, are included. The model is then executed to obtain predictions, and we apply `LayerIntegratedGradients` from the Captum library to compute attribution scores relative to the model's embedding layer [7]. We isolate the test code segment using a regular expression pattern that spans from the `@Test` annotation to the closing brace, and we compute IG scores only for that region. We normalize the resulting attribution scores and filter out stopwords. Finally, we rank the remaining tokens based on their importance to highlight which parts of the code most influence the model's decision.

## 6.4 Baselines

We compare against two other LLM-based flaky test classifiers, Flakify [20] and FlakyCat [13]. Flakify predicts whether a test is flaky or not through a fine-tuned CodeBERT model, built using training from an existing dataset of flaky tests [2]. Since we want to classify tests into one of six different categories as opposed to just doing a binary classification (flaky vs non-flaky), we adapt the existing Flakify code [8] to perform multi-class classification over our six categories. We mainly modify the final classification layer of the CodeBERT-based Flakify model to include a linear head outputting logits for each of our six categories instead of just two. Then we retrain the model using our labeled dataset.

FlakyCat, on the other hand, was already designed to classify flaky tests into flaky test categories [13]. In the original work, it classifies flaky tests into one of four flaky test categories based on an existing dataset, with tests already labeled with the flaky test categories. However, FlakyCat does not fine-tune the pre-trained CodeBERT model; instead, it extracts frozen embeddings from CodeBERT and trains a lightweight classification head on top. We directly use the existing FlakyCat code [9].

For both Flakify and FlakyCat, to ensure proper comparison against FlakyLens, while we reuse their existing code as much as possible, we retrain new models using the existing code to construct new models using the same dataset we use for training FlakyLens. We use the same training sets for all three flaky test classifiers, and we evaluate on the same test sets, to ensure fair comparisons between the flaky test classifiers.

## 6.5 Measurement Metrics

A flaky test classifier can predict a test into several different flaky test categories, and we consider each flaky test category separately when calculating the evaluation metrics. For each flaky test category, we define true positives (TP) as tests correctly predicted to belong to that category, false positives (FP) as tests incorrectly predicted to belong to that category, and false negatives (FN) as tests of the category that were misclassified as another. We then compute precision, recall, and F1-score for each flaky test category individually. We compute precision as $\frac{TP}{TP+FP}$, recall as $\frac{TP}{TP+FN}$, and F1-score as $2 \times \frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$.

To aggregate these metrics across all flaky test categories, we use the macro-average, which computes the unweighted mean of the per-category scores, treating all categories equally. Specifically, the macro F1-score is calculated by averaging the F1-score of each category (C), ensuring that minority categories are not overshadowed by majority categories, $\frac{1}{C}\sum_{i=1}^{C}\text{F1-score}_i$. We obtain these metrics using the *classification report* from sklearn [6] and report these as the final results in Section 7.

## 6.6 Hardware Environment

We perform fine-tuning on a Linux machine equipped with a single NVIDIA RTX A5000 GPU, 125GB of RAM, and 48GB of GPU memory, utilizing CUDA version 12.0. The same setup is used for running the Huggingface open-source models to perform flaky-test classification and compute gradient attributions.

## 7 Evaluation

### 7.1 RQ1: FlakyLens vs Prior Flaky Test Classifiers

Table 2 presents the results of FlakyLens compared to other LLM-based flaky test classifiers such as Flakify (both the data-leakage and non-data-leakage versions) and FlakyCat. The results show each classifier's F1-score for each category: Async Wait (Async.), Concurrency (Conc.), Time (Time), Unordered Collections (UC), Test Order Dependency (OD), and non-flaky (Non-flaky). We see that the original Flakify indeed performs abnormally well due to data leakage issue during training (row "Flakify (Original)"), with a big drop after the issue is addressed (row "Flakify (With Fix)"). When we compare the results of FlakyLens against the corrected Flakify version, we see that FlakyLens improves the F1-score by more than 9.17 percentage points (pp), based on the macro-average. Compared to FlakyCat, FlakyLens achieves an improvement of 13.79pp.

Table 2. F1-score comparison of different techniques across various flaky test categories. The columns represent the different flaky test categories (Async., Conc., Time, UC, OD) and non-flaky tests. The last column (Macro Avg.) shows the overall macro-average. First four rows show results for different flaky test classifiers, including FlakyLens. Last two rows show improvement of FlakyLens over Flakify and FlakyCat.

| | Category | | | | | | Macro |
|---|---|---|---|---|---|---|---|
| Technique | Async. | Conc. | Time | UC | OD | Non-flaky | Avg. |
| Flakify (Original) | 73.37 | 64.65 | 82.09 | 83.41 | 88.53 | 100.00 | 81.82 |
| Flakify (With Fix) | 42.80 | 29.62 | 72.55 | 39.15 | 54.70 | **100.00** | 56.62 |
| FlakyCat | 43.75 | 20.00 | 53.75 | 58.75 | 36.25 | 99.50 | 52.00 |
| FlakyLens (Ours) | **58.37** | **35.92** | **72.73** | **73.63** | **64.35** | **100.00** | **65.79** |
| ↑ over Flakify | 15.57 | 6.30 | 0.18 | 34.48 | 9.65 | 0.00 | 9.17 |
| ↑ over FlakyCat | 14.62 | 15.92 | 18.98 | 14.88 | 28.10 | 0.50 | 13.79 |

We observe that all techniques correctly predict the non-flaky category very well, achieving an F1-score close to 100%, likely due to there being so many more non-flaky tests. FlakyLens achieves the highest improvements in F1-score specifically for the Unordered Collections category over the other baselines, achieving an improvements of 34.48pp and 14.88pp over Flakify and FlakyCat, respsectively. Interestingly, while all techniques struggle the most with the Concurrency category, FlakyLens still outperforms both baselines for this category. Specifically, it achieves the highest F1-score of 35.92%, exceeding Flakify and FlakyCat by 6.30pp and 15.92pp, respectively. These results demonstrate that FlakyLens consistently outperforms state-of-the-art approaches across all flaky test categories.

To better understand the misclassifications, we analyze which categories the incorrectly predicted tests are classified into. Figure 8 illustrates how tests from each ground truth category are misclassified into other categories. The horizontal axis represents the predicted category, while the vertical axis indicates the actual category. Each cell displays the number of tests misclassified into the corresponding category.

Overall, the model misclassifies a total of 106 tests. The highest number of misclassifications (40 out of 93) occurs in the Test Order Dependency category (Category 4), with most of these tests incorrectly predicted as Async Wait (Category 0). However, the highest misclassification rate is observed in the Concurrency category (Category 1), where 25 out of 37 tests, accounting for 67.6%, are misclassified, predominantly as Async Wait. This result suggests that the distinction between Concurrency and Async Wait is subtle, likely due to overlapping characteristics in timing and

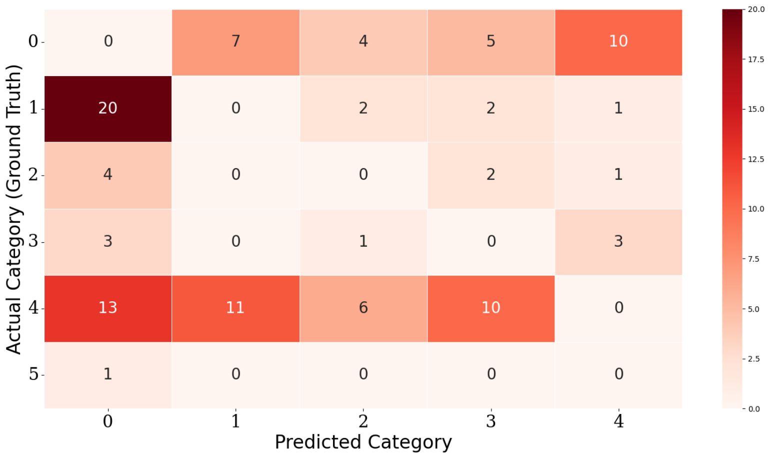


Fig. 8. Breakdown of the predictions of the 106 tests by FlakyLens.

thread-related behaviors. Incorporating deeper program analysis or runtime information could help better differentiate between these categories. In contrast, the non-flaky category (Category 5) sees the fewest misclassifications, with only one case that was misclassified as Async Wait.

***RQ1:** FlakyLens outperforms prior flaky test classifiers Flakify and FlakyCat at predicting different flaky test categories. It improves the F1-score by over 9.17pp compared to Flakify (after fixing the data-leaking issue) and by 13.79pp compared to FlakyCat.*

## 7.2 RQ2: FlakyLens vs State-of-the-Art Generative LLMs

We compare the performance of FlakyLens against the latest open-source generative LLMs, including Qwen2-7B, LLAMA-3-8B, DS-Coder-16B, CodeLlama-13B, Gemma-7b, and CodeGemma-7B for flaky-test classification task. These models are much larger and pre-trained on Internet-scale data. Hence, we utilize them in a zero-shot setting.

Table 3 presents the F1-score for each category as predicted by the LLMs. The results show that, based on macro-average, Qwen2-7B performs the best among these pre-trained LLMs, achieving an F1-score of 14.86%, followed by LLAMA-3-8B, which achieves 13.57%. Interestingly, these two models have an F1-score of 0.00% for the Test Order Dependency (OD) category, indicating that none of the tests in this category were correctly predicted. On the other hand, DS-Coder-16B achieves the highest F1-score for the Test Order Dependency category. Additionally, Table 3 reveals that CodeGemma-7B performs the worst, achieving an F1-score of 0.64%, making it the least effective model among those evaluated. DS-Coder-16B achieves the best performance compared to all other models across all flaky test categories, only performing poorly for the non-flaky category. While it outperforms the other models at identifying the correct category for actual flaky tests, it still performs worse than FlakyLens for all flaky test categories (also shown in the table).

While F1-score provides an overall measure of performance, it does not show the actual number of correctly predicted tests, which is also crucial for understanding model effectiveness. We further analyze the absolute number of correctly predicted tests by each model, as shown in Figure 9 and Figure 10. Figure 9 presents the total number of tests whose categories are correctly predicted by these models, including FlakyLens. We observe that out of 8574 tests, FlakyLens correctly predicts

Table 3. Evaluation of open-source LLM models for flaky test category prediction using F1-score. All models used are the instruct versions of the LLMs. DS-Coder-16B represents DeepSeek-Coder V2 Lite.

| LLM Model | Categories | | | | | | Macro |
|---|---|---|---|---|---|---|---|
| | Async. | Conc. | Time | UC | OD | Non-flaky | Avg. |
| Qwen2-7B | 7.41 | 5.30 | 9.76 | 4.12 | 0.00 | **80.15** | **14.86** |
| LLAMA-3-8B | 5.09 | 8.03 | 11.45 | 2.63 | 0.00 | 71.84 | 13.57 |
| DS-Coder-16B | **21.23** | **16.59** | **16.55** | **4.86** | **16.86** | 0.37 | 9.61 |
| CodeLlama-13B | 2.12 | 1.74 | 10.41 | 1.26 | 2.69 | 0.00 | 2.18 |
| Gemma-7b | 3.33 | 2.14 | 0.00 | 0.00 | 0.92 | 0.00 | 0.79 |
| CodeGemma-7B | 1.39 | 5.68 | 0.24 | 0.27 | 0.56 | 0.00 | 0.64 |
| FlakyLens (Ours) | 58.37 | 35.92 | 72.73 | 73.63 | 64.35 | 100.00 | 65.79 |

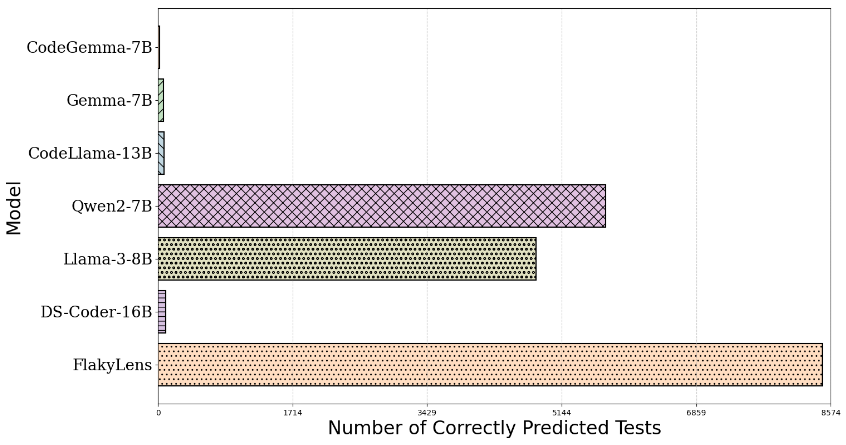


Fig. 9. Number of correctly predicted flaky tests by each model.

the category for 8468 tests, significantly outperforming the second-best model, Qwen2-7B, which achieves 5703 correct predictions. At the lower end, CodeGemma-7B correctly predicts the category for only 19 tests. However, due to the presence of a large number of non-flaky tests, the absolute count of correctly predicted tests does not fully illustrate how well each model classifies flaky tests. To address this issue, Figure 10 shows only the number of correctly predicted flaky tests. Here, we again see that FlakyLens outperforms the other models, correctly predicting the catogory of 175 flaky tests.

Beyond simply observing the absolute number of correctly predicted tests, we also want to know whether different models are making correct predictions for the same tests or different ones. Figure 11 and Figure 12, are heatmaps illustrating the percentage of tests correctly predicted by each pair of models. From these figures, we observe that the highest agreement occurs between FlakyLens and Qwen2-7B, with 66.22% of tests predicted correctly by both models. However, once again due to the large proportion of non-flaky tests, this visualization does not provide a clear understanding of how well the models identify flaky test categories. To further refine this analysis, Figure 12 focuses exclusively on flaky tests, highlighting both the number of correctly predicted flaky tests and the correlation between different LLMs in identifying them. This figure reveals that

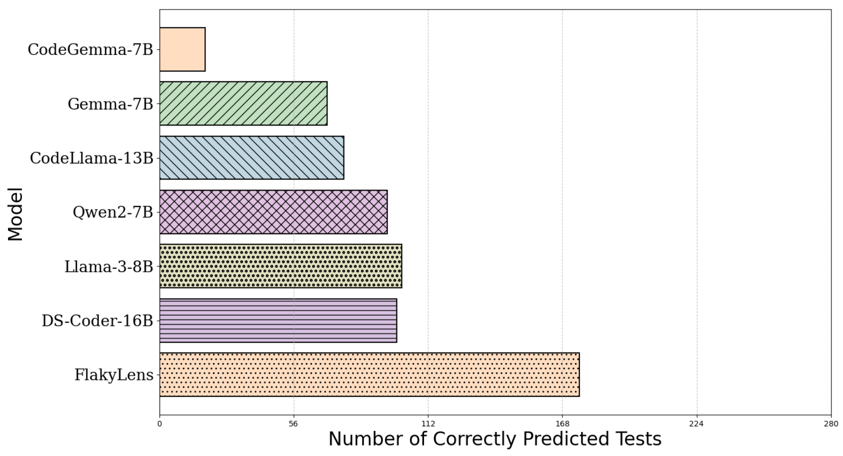
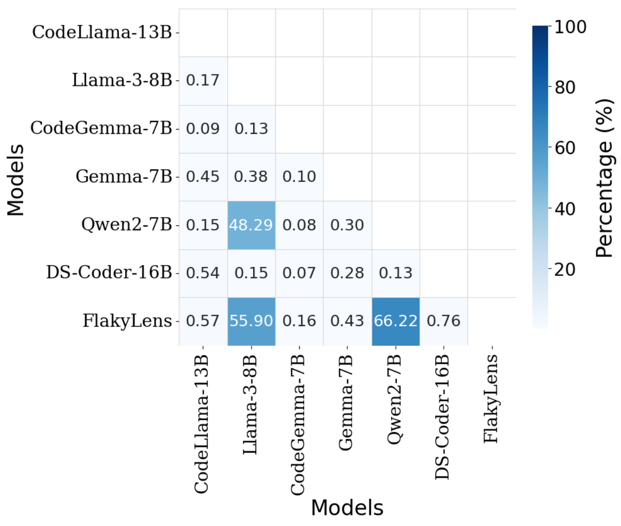


Fig. 10. Number of correctly predicted flaky tests by each model, excluding the non-flaky category.



|  | CodeLlama-13B | Llama-3-8B | CodeGemma-7B | Gemma-7B | Qwen2-7B | DS-Coder-16B | FlakyLens |
|---|---|---|---|---|---|---|---|
| CodeLlama-13B |  |  |  |  |  |  |  |
| Llama-3-8B | 0.17 |  |  |  |  |  |  |
| CodeGemma-7B | 0.09 | 0.13 |  |  |  |  |  |
| Gemma-7B | 0.45 | 0.38 | 0.10 |  |  |  |  |
| Qwen2-7B | 0.15 | 48.29 | 0.08 | 0.30 |  |  |  |
| DS-Coder-16B | 0.54 | 0.15 | 0.07 | 0.28 | 0.13 |  |  |
| FlakyLens | 0.57 | 55.90 | 0.16 | 0.43 | 66.22 | 0.76 |  |

Fig. 11. Pairwise agreement among models, showing the percentage of test cases correctly predicted by both models. Higher percentages indicate stronger prediction overlap. FlakyLens and Qwen2-7B exhibit the highest agreement.

the highest overlap in correctly predicted flaky tests occurs between Llama-3-8B and Qwen2-7B, reaching 28.93%.

***RQ2:*** *FlakyLens significantly outperforms open-source LLMs in flaky test classification, achieving far higher F1-scores and correct predictions, while general-purpose models such as Qwen2-7B, Llama-3-8B struggle, especially with flaky test categories like Test Order Dependency (OD).*

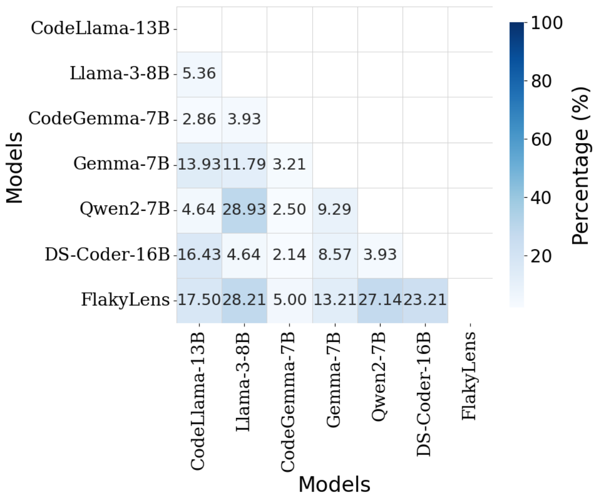


Fig. 12. Pairwise agreement among models (excluding tests from the non-flaky category), showing the percentage of test cases correctly predicted by both models. Higher percentages reflect stronger prediction overlap. The highest agreement is observed between Qwen2-7B and Llama-3-8B, followed by FlakyLens and Llama-3-8B.

Table 4. Most and least important tokens per category.

| ID | Top-5 Most Imp. | | Top-5 Least Imp. | |
|---|---|---|---|---|
| | Tokens | Attribution | Tokens | Attribution |
| 1 | sleep;topic;wait;get;Interrupted | 1.252–0.853 | crashed;Meta;Asserted;Connected;Upgrade | 0.002–0.001 |
| 2 | Duration;new;Interrupted;verify;Subscriber | 0.825–0.115 | cluster;Tokens;Mode;Mill;List | 0.002–0.002 |
| 3 | Time;Network;when;Run;wait | 0.735–0.361 | true;maximum;compare;STACKTRACE;left | 0.003–0.002 |
| 4 | List;Equals;Enum;JSON;set | 0.563–0.121 | Partition;parse;Fast;Value;ependencies | 0.002–0.000 |
| 5 | Permission;naming;Action;Name;Composite | 3.743–0.565 | ubbo;Remoting;All;No;null | -0.000 – -0.001 |
| 6 | Exception;get;That;Equals;expected | 41.09–20.34 | Executes;somekey;Violation;Course;izard | -0.000 – -0.000 |

## 7.3 RQ3: Important Tokens

Table 4 presents the Top-5 most and Top-5 least important tokens for each flaky test category, based on token weighted attribution scores. For example, in the Async Wait category, tokens such as “sleep”, “topic”, “wait”, “get”, and “Interrupted” consistently appear in correctly classified tests, namely 17, 4, 5, 10, and 4 times out of 50, respectively. In the Concurrency category, “Duration” and “new” are the top indicators, occurring in 1 and 4 out of 12 tests, respectively. We also show in the table the range of weighted attribution scores across the five tokens shown for each category.

Similar patterns are found in other categories: “Time” and “Network” are highly relevant tokens for the Time category, and data structure terms like “List”, “Equals”, “Enum” and “JSON” are most important for the Unordered Collections category. For the Test Order Dependency category, tokens such as “Permission”, “naming”, and “Action” dominate, while for non-flaky tests, “Exception” and “get” appear most frequently (1375 and 1152 times).

In contrast, the least important tokens, which contribute minimally to the model’s classification based on weighted attribution score include generic, utility-related, or syntactically common terms such as “Upgrade”, “Connected”, “cluster”, or “true”. These tokens appear across many tests but do not meaningfully help distinguish the correct flaky test category. By identifying these tokens with low weighted attribution scores, we help highlight which code elements are not useful for classification in contrast to the most important tokens.

Our findings highlight which code elements signal different types of flaky behavior to the model, offering transparency into what drives classification decisions. These results can help developers gain intuition about what characteristics make tests flaky, potentially guiding better test design and debugging. Additionally, understanding token importance can be useful in interpreting model decisions and building trust in automated flaky test detection tools.

> ***RQ3:*** *Token attribution highlights key linguistic and structural patterns behind flaky test classification, such as the significance of "sleep", "Duration", and "new" for the Async Wait and Concurrency categories. These insights not only explain model decisions but also can help developers reason about why their test is flaky.*

## 7.4 RQ4: Impact of Important Tokens via Adversarial Perturbation

Table 5 shows the impact of important tokens when used as perturbation in test code, without altering the original test's semantics. The goal is to determine whether injecting code that contains these most important tokens, even without changing semantics, can influence the model's prediction. We apply our five different perturbation techniques (Section 5.2), each applied individually per test, across the six test categories.

Each column in the table corresponds to a specific perturbation technique and is split into two sub-columns: the first shows the resulting F1-score after applying that type of perturbation, and the second (∇) indicates the change in F1-score relative to the original (clean) version, reported in percentage points (pp). A larger negative value indicates a greater performance drop due to the applied perturbation. For example, in the Async Wait category, the model originally achieves an F1-score of 58.37%. However, after applying the deadcode perturbation, the F1-score drops by -9.64pp, resulting in a new F1-score of 48.72%.

Table 5. Impact on prediction when we considered *Most* important tokens for perturbation.

| | Original | Perturbation | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Deadcode | | Print | | Variable | | Multi-Line | | Single-Line | |
| ID | F1 | F1 | ∇ | F1 | ∇ | F1 | ∇ | F1 | ∇ | F1 | ∇ |
| 1 | 58.37 | 48.72 | -9.64 | 50.68 | -7.68 | 49.25 | -9.12 | 54.63 | -3.74 | 49.46 | -8.91 |
| 2 | 35.92 | 25.81 | -10.11 | 30.32 | -5.59 | 32.84 | -3.08 | 35.65 | -0.27 | 32.97 | -2.95 |
| 3 | 72.73 | 45.45 | -27.27 | 55.79 | -16.94 | 51.61 | -21.12 | 59.58 | -13.15 | 51.85 | -20.88 |
| 4 | 73.63 | 35.15 | -38.49 | 59.78 | -13.85 | 53.98 | -19.66 | 49.46 | -24.17 | 56.10 | -17.54 |
| 5 | 64.35 | 39.66 | -24.70 | 58.48 | -5.87 | 51.99 | -12.37 | 42.91 | -21.44 | 41.55 | -22.81 |
| 6 | 100.00 | 100.00 | 0.00 | 100.00 | 0.00 | 99.48 | -0.52 | 100.00 | 0.00 | 100.00 | 0.00 |
| **Avg. ∇** | - | - | **-18.37** | - | **-8.32** | - | **-10.98** | - | **-10.46** | - | **-12.18** |

Overall, we observe that the maximum average F1-score degradation occurs when using the deadcode perturbation technique, where the average F1-score degradation is -18.37pp. The second highest average degradation is caused by the single-line comment perturbation (-12.18pp).

Similarly, we also present the impact of the least important tokens based on weighted attribution scores in Table 6. In the table, we see that the deadcode perturbation technique still results in the higher loss in F1-score (-7.91pp), though this loss is much smaller than the loss caused with the deadcode perturbation technique using the most important tokens (which caused the least

Table 6. Impact on prediction when we use *Least* important tokens for perturbation.

| | Original | Perturbation | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Deadcode | | Print | | Variable | | Multi-Line | | Single-Line | |
| ID | F1 | F1 | ∇ | F1 | ∇ | F1 | ∇ | F1 | ∇ | F1 | ∇ |
| 1 | 58.37 | 50.38 | -7.99 | 57.76 | -0.61 | 52.53 | -5.84 | 50.38 | -7.99 | 55.78 | -2.59 |
| 2 | 35.92 | 29.05 | -6.86 | 33.84 | -2.08 | 33.08 | -2.84 | 29.05 | -6.86 | 34.16 | -1.76 |
| 3 | 72.73 | 69.82 | -2.91 | 71.09 | -1.64 | 65.79 | -6.94 | 69.82 | -2.91 | 68.39 | -4.33 |
| 4 | 73.63 | 63.07 | -10.56 | 69.93 | -3.71 | 64.71 | -8.93 | 63.07 | -10.56 | 66.32 | -7.32 |
| 5 | 64.35 | 40.46 | -23.89 | 61.85 | -2.51 | 58.00 | -6.35 | 40.46 | -23.89 | 58.87 | -5.48 |
| 6 | 100.00 | 100.00 | 0.00 | 100.00 | 0.00 | 99.48 | -0.52 | 100.00 | 0.00 | 100.00 | 0.00 |
| **Avg. ∇** | - | - | **-7.91** | - | **-1.76** | - | **-5.24** | - | **-8.70** | - | **-3.58** |

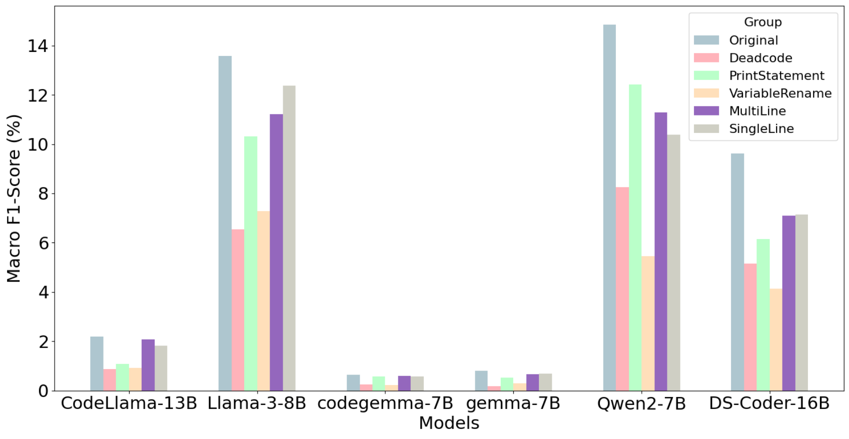


Fig. 13. Result of different LLMs when apply different perturbations in the data.

change in F1-score among all techniques when using the most important tokens). Similarly, all other perturbation techniques have a smaller impact on the performance of the models: the F1-score drops ranging from -8.70pp to -1.76pp, all less than that from perturbation techniques using the most important tokens.

Figure 13 illustrates how the performance of open-source LLMs degrades when evaluated on tests with added perturbation. Each bar represents a model's macro-average F1-score under different perturbation techniques. In all cases, performance on perturbed tests is consistently lower than on the original, unmodified tests, highlighting the critical role of the identified important tokens based on weighted attribution score, even though we are using important tokens obtained when analyzing a completely different model.

**Retraining using Perturbed Test Code**. It is interesting to note that FlakyLens could be retrained using perturbed test code examples, to see whether it can be made more robust to these types of semantics-preserving perturbations. To evaluate the effects of retraining, we augment each training set we used in our evaluation with additional tests where we applied our perturbation techniques on each test. For a training set, we randomly sample 20% of the tests, and for each test, we randomly choose another category that the test is not and apply each of the five perturbation techniques using important tokens from that chosen other category. We therefore generate five new perturbed versions of the test and add them into the training set, effectively doubling the overall size of the training set. We retrain the flaky test classifier with the new training set per fold, and then we re-evaluate the flaky test classifiers of each fold on the same corresponding test set as we did for RQ4. We find that the newly trained flaky test classifiers have a much smaller loss in F1-score. However, while these flaky test classifiers perform well now on the perturbed tests, we find that there is very little change in how well they classify the non-perturbed tests. In the future, we plan to investigate new perturbation techniques that can be also used in this manner to help train more robust flaky test classifiers, obtaining better performance on real-world flaky tests.

***RQ4:*** *Perturbing tests with important tokens based on weighted attribution score, without changing semantics, substantially degrades model performance, revealing a strong reliance on these tokens for accurate classification. In contrast, using the least important tokens has minimal impact on prediction accuracy, suggesting the impact is not only due to the act of perturbation.*

## 8 Threats to Validity

*External Validity.* Our study focuses on open-source Java projects, which may limit the generalizability of the results to other programming languages. While Java is widely used, the effectiveness of FlakyLens may vary when applied to projects written in other languages with different testing frameworks, paradigms, or architectures (e.g., Python or C++). However, the core principles and methodology of FlakyLens are language-agnostic. It would be interesting future work to evaluate on different languages and see whether the overall findings themselves can be language-agnostic.

We compare FlakyLens against existing open-source LLMs in a zero-shot setting, using just prompting to predict flaky test categories. We do not use the much larger models such as GPT-4.1 or Claude 3.7 Sonnet due to the large costs associated with using these models when run on a large number of tests in our dataset. It would be interesting future work to also evaluate against such models.

Our study focuses on a well-defined flaky test categories, such as Async Wait, Concurrency, Time, and others, collected from prior work [13] and commonly observed in practice. While real-world test suites may have many other flakiness causes, for example environment-specific behaviors, network delays, or GUI-related issues that are not used in this study, the ones we have in our dataset tend to be the most prevalent categories as found in past work [34].

*Internal Validity.* Data leakage between the training and evaluation sets is a common threat that can compromise the validity of results. Tests from the same project may sometimes share structural or contextual similarities, and including them across both splits could artificially inflate model performance. To mitigate this problem, we adopt a strict per-project evaluation strategy: if any test from a project is used during training, we ensure that no other test from that project appears in the evaluation set. This setup prevents both direct and indirect information leakage and ensures a more realistic assessment of generalization performance.

We train and evaluate the model using the same dataset, and our flaky test classifier achieves performance consistent with previously reported results, providing confidence in the correctness of our implementation. While the model is trained to predict flaky test categories, some degree of non-determinism may still occur. To address this problem, we set a fixed random seed for all experiments. Additionally, we run the evaluation script 10 times to ensure the model produces consistent and deterministic outputs.

FlakyLens only uses test code to predict the flaky test categories. If the source of flakiness comes from somewhere outside of test code, it is possible that FlakyLens would be unable to reason properly about that type of flakiness. In the future, expanding FlakyLens's scope to handle these external setups could address such issues.

*Construct Validity.* In this work, we leverage previously labeled flaky test categories established through prior research and domain expertise. These labels provide a solid foundation for training and evaluating our models. While some flaky test categories, such as Async Wait and Concurrency, may naturally share overlapping behaviors, the use of well-informed annotations allows us to train models that reflect real-world distinctions. We acknowledge that further enhancement such as expanding the number of labeled instances per category, could enhance classification accuracy.

## 9 Related Work

### 9.1 Studies on Flaky Tests

Luo et al. was the first to perform an empirical study on flaky tests in open-source projects [34]. They categorized flaky tests by inspecting developer-fixed flaky tests manually. Later researchers would develop techniques to automatically detect flaky tests, guided by the results from this empirical study [23, 29, 30, 44, 47]. For example, Lam et al. proposed iDFlakies that reruns tests in different orders to detect order-dependent flaky tests [29], and Shi et al. developed NonDex to detect tests that assume deterministic implementations of nondeterministic specifications, such as assuming deterministic ordering of unordered collections [44]. These two types of tests are represented as the Test Order Dependency and Unordered Collections flaky test categories in our work.

### 9.2 Flaky Test Classification

Many flaky-test detection techniques rely on rerunning tests or dynamic analysis, meaning they can be costly to run. Prior work investigated machine learning (ML) techniques to predict whether a test could be flaky. Alshammari et al. proposed FlakeFlagger, the first ML-based technique for detecting flaky tests [14]. They created a large dataset of flaky tests by rerunning tests 10,000 times, and then they used this dataset to train and evaluate their ML classifier at predicting whether a test is flaky. They train their classifier to use static features like test smells and lines of code as well as dynamic features like coverage and test runtime.

Fatima et al. later proposed Flakify, which uses LLMs to similarly predict test flakiness, fine-tuning the pre-trained CodeBERT model to extract features from just test code. They find improved performance, with a reported F1-score of 98% on their dataset. In our work, we train FlakyLens similarly to Flakify, by fine-tuning CodeBERT, but we introduce a classification head that can predict different flaky test categories, not just a binary classification of flaky vs non-flaky. We also make additional changes to training parameters to take into consideration the imbalance of flaky vs non-flaky tests, representative of real-world flaky test distributions.

Akli et al. proposed FlakyCat for classifying flaky tests into different flaky test categories. They obtained a dataset of flaky tests labeled by the reason for flakiness [15], based on the definitions provided by Luo et al. [34]. FlakyCat also leverages CodeBERT to extract features from test code, and then it later uses few-shot learning to train a classifier to predict the flaky test category. They similarly evaluate using some traditional ML classifiers that use CodeBERT features for doing the same prediction, finding that FlakyCat performs better than those traditional ML classifiers. FlakyLens has the same goal as FlakyCat, so we compare against it as a baseline, though we fine-tune FlakyLens for the task instead of using few-shot learning.

Later, Rahman et al. proposed Q-Flakify++ to similarly categorize flaky tests but emphasizing reduced prediction time via quantization [40]. Their quantized model also demonstrates noteworthy F1-score. Note that Q-Flakify++ is a quantized version of Flakify adjusted to categorize flaky tests, finding that quantization results in a slight loss in accuracy. As such, we only compare against

Flakify and not Q-Flakify++. Ultimately, for all these LLM-based flaky test classifiers, it remains unclear *why* they have such high F1-scores and why they rely on to make decisions. Therefore, we focus on extracting the important tokens from tests and evaluating their impact on prediction by leveraging perturbation techniques that use those important tokens.

### 9.3 Adversarial Attacks on Code LLMs

Yang et al. [48] proposed ALERT, which generates adversarial attacks for code LLMs by applying perturbations that appear natural to humans (e.g., variable names that are similar in meaning). Jha et al.[25] developed CodeAttack, a black-box method for adversarial attacks for code language models, and show that it leads to drops in performance on many code-code (translation and repair) and code-NL (summarization) tasks. More recently, Gao et al. [21] proposed a general attack and defense mechanism for models of code, which significantly improved over the prior techniques. While our work uses similar perturbations on code LLMs, we leverage domain-specific tokens that are more suited to attacking flaky test classifiers.

## 10 Conclusions

In this work, we show LLM-based flaky test classifiers can be lacking in accuracy, in contrast to prior results, suggesting it to be a challenging task for LLMs. We propose FlakyLens that utilizes a different strategy for fine-tuning, resulting in an improved flaky test classifier. We also build FlakeBench, a labeled dataset of flaky tests with a more representative mix of both flaky and non-flaky tests.

We also identified important tokens based on attribution scores. We confirm that they impact flaky test classifier performance, with many more mispredictions when run on test code with perturbations based on these important tokens, demonstrating their impact on the decision-making. We also compare against large pre-trained models instructed to perform the same task in a zero-shot setting, but they do not perform as well as FlakyLens. Our work underscores the need for enhanced training strategies to improve flaky test classifiers.

In the future, we plan to expand FlakeBench by adding more flaky test categories and including projects in different programming languages. These changes can further enhance the robustness and generalizability of classifiers across more diverse settings.

## Data Availability

Our artifact for reproducing results is publicly available [11]. We present additional results and experiment information on our website [12]. Our code is open-source and available on GitHub [10].

## Acknowledgements

This work was partially funded by National Science Foundation grants CCF-2145774 and CCF-2217696.

## References

[1] 2016. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.
[2] 2021. IDoFT. http://mir.cs.illinois.edu/flakytests.
[3] 2021. netty. https://github.com/netty/netty.
[4] 2023. Code Pretraining Models. https://github.com/microsoft/CodeBERT.
[5] 2024. huggingface. https://huggingface.co/.
[6] 2024. sklearn. https://scikit-learn.org/1.5/modules/generated/sklearn.metrics.classification_report.html.
[7] 2025. Captum. https://captum.ai/docs/extension/integrated_gradients.
[8] 2025. Flakify. https://github.com/uOttawa-Nanda-Lab/Flakify/commit/3b726f44d5ce5dffc9b190327bc0b95d557c5c6a.

[9] 2025. FlakyCat. https://github.com/Amal-AK/FLAKYCAT.

[10] 2025. FlakyLens. https://github.com/UT-SE-Research/FlakyLens.

[11] 2025. Understanding and Improving Flaky Test Classification Artifact. doi:10.5281/zenodo.15761937

[12] 2025. Understanding and Improving Flaky Test Classification Website. https://sites.google.com/view/robust-model.

[13] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2023. FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. In *ACM/IEEE International Conference on Automation of Software Test*. 140–151. doi:10.1109/AST58925.2023.00018

[14] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *International Conference on Software Engineering*. 1572–1584. doi:10.1109/ICSE43902.2021.00140

[15] Keila Barbosa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda. 2023. Test Flakiness Across Programming Languages. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2039–2052. doi:10.1109/TSE.2022.3208864

[16] Saikat Dutta, Jeeva Selvam, Aryaman Jain, and Sasa Misailovic. 2021. TERA: optimizing stochastic regression tests in machine learning projects. In *International Symposium on Software Testing and Analysis*. 413–426. doi:10.1145/3460319.3464844

[17] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Misailovic Sasa. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *International Symposium on Software Testing and Analysis*. 211–224. doi:10.1145/3395363.3397366

[18] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. FLEX: Fixing Flaky Tests in Machine-Learning Projects by Updating Assertion Bounds. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 603–614. doi:10.1145/3468264.3468615

[19] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840. doi:10.1145/3338906.3338945

[20] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1912–1927. doi:10.1109/TSE.2022.3201209

[21] Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete adversarial attack to models of code. In *Conference on Programming Language Design and Implementation*. 172–195. doi:10.1145/3591227

[22] Martin Gruber, Stephan Lukasczyk, Florian Krois, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *International Conference on Software Testing, Verification, and Validation*. 148–158. doi:10.1109/ICST49551.2021.00026

[23] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*. 223–233. doi:10.1145/2771783.2771793

[24] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. 2021. PyTorch. *Programming with TensorFlow: Solution for Edge Computing Applications* (2021), 87–104.

[25] Akshita Jha and Chandan K. Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 14892–14900. doi:10.1609/aaai.v37i12.26739

[26] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. 2020. Captum: A unified and generic model interpretability library for pytorch. *arXiv preprint arXiv:2009.07896* (2020). doi:10.48550/arXiv.2009.07896

[27] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *International Conference on Software Engineering, Software Engineering in Practice*. 110–119. doi:10.1145/3377813.3381370

[28] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *International Symposium on Software Testing and Analysis*. 101–111.

[29] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*. 312–322. doi:10.1109/ICST.2019.00038

[30] 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 *International Symposium on Software Reliability Engineering*. 403–413. doi:10.1109/ISSRE5003.2020.00045

[31] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1520. doi:10.1145/3540250.3558956

[32] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *International Conference on Computer Vision*. 2980–2988. doi:10.1109/ICCV.2017.324

[33] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *International Conference on Neural Information Processing Systems*. 4768–4777. doi:10.5555/3295222.3295230

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653. doi:10.1145/2635868.2635920

[35] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *International Conference on Software Engineering, Software Engineering in Practice*. 91–100. doi:10.1109/ICSE-SEIP.2019.00018

[36] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*. 233–242. doi:10.1109/ICSE-SEIP.2017.16

[37] John Micco. 2017. The state of continuous integration testing@ google. In *International Conference on Software Testing, Verification, and Validation*.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *International Conference on Neural Information Processing Systems*. doi:10.5555/3454287.3455008

[39] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *Mining Software Repositories*. 492–502.

[40] Shanto Rahman, Abdelrahman Baz, Sasa Misailovic, and August Shi. 2024. Quantizing Large-Language Models for Predicting Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. 93–104. doi:10.1109/ICST60714.2024.00018

[41] Shanto Rahman, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. 2024. Automatically Reproducing Timing-Dependent Flaky-Test Failures. In *International Conference on Software Testing, Verification, and Validation*. 269–280. doi:10.1109/ICST60714.2024.00032

[42] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. In *International Conference on Knowledge Discovery and Data Mining*. 1135–1144. doi:10.1145/2939672.2939778

[43] Soumya Sanyal and Xiang Ren. 2021. Discretized integrated gradients for explaining language models. *arXiv preprint arXiv:2108.13654* (2021). doi:10.48550/arXiv.2108.13654

[44] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *International Conference on Software Testing, Verification, and Validation*. 80–90. doi:10.1109/ICST.2016.40

[45] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *International Conference on Software Maintenance and Evolution*. 301–311. doi:10.1109/ICSME46990.2020.00037

[46] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*. 3319–3328. doi:10.5555/3305890.3306024

[47] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*. 120–124. doi:10.1145/3510454.3516846

[48] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493. doi:10.1145/3510003.3510146

[49] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*. 385–396. doi:10.1145/2610384.2610404

[50] Zhenxun Zhuang, Mingrui Liu, Ashok Cutkosky, and Francesco Orabona. 2022. Understanding AdamW through Proximal Methods and Scale-Freeness. *Transactions on Machine Learning Research* (2022). doi:10.48550/arXiv.2202.00089

[51] Celal Ziftci and Jim Reardon. 2017. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale. In *International Conference on Software Engineering*. 113–122. doi:10.1109/ICSE-SEIP.2017.13

Received 2025-03-26; accepted 2025-08-12