FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning

Amal Akli
University of Luxembourg
Luxembourg
amal.akli@uni.lu

Guillaume Haben
University of Luxembourg
Luxembourg
guillaume.haben@uni.lu

Sarra Habchi
Ubisoft
Canada
sarra.habchi@ubisoft.com

Mike Papadakis
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

***Abstract*—Flaky tests are tests that yield different outcomes when run on the same version of a program. This non-deterministic behaviour plagues continuous integration with false signals, wasting developers' time and reducing their trust in test suites. Studies highlighted the importance of keeping tests flakiness-free. Recently, the research community has been pushing towards the detection of flaky tests by suggesting many static and dynamic approaches. While promising, those approaches mainly focus on classifying tests as flaky or not and, even when high performances are reported, it remains challenging to understand the cause of flakiness. This part is crucial for researchers and developers that aim to fix it. To help with the comprehension of a given flaky test, we propose FlakyCat, the first approach to classify flaky tests based on their root cause category. FlakyCat relies on CodeBERT for code representation and leverages Siamese networks to train a multi-class classifier. We train and evaluate FlakyCat on a set of 451 flaky tests collected from open-source Java projects. Our evaluation shows that FlakyCat categorises flaky tests accurately, with an F1 score of 73%. Furthermore, we investigate the performance of our approach for each category, revealing that *Async waits*, *Unordered collections* and *Time*-related flaky tests are accurately classified, while *Concurrency*-related flaky tests are more challenging to predict. Finally, to facilitate the comprehension of FlakyCat's predictions, we present a new technique for CodeBERT-based model interpretability that highlights code statements influencing the categorization.**

***Index Terms*—Software Testing, Flaky Tests, CodeBERT, Few-Shot learning, Siamese Networks.**

I. Introduction

Continuous Integration (CI) plays a key role in nowadays software development life cycle [1], [2]. CI ensures the quick application of changes to a main code base by automatically running a variety of tasks. Those changes are responsible for building the program and its dependencies, performing checks (*e.g.*, static analysis), and running test suites to maintain code integrity and correctness. An important assumption for practitioners is that tasks are deterministic, *i.e.*, regardless of the execution's context of a same task, results need to remain similar.

Unfortunately, in practice, this is not always the case. Previous research has identified test flakiness as one of the main issues in the application of automated software testing [3]–[5]. A flaky test is a test that passes and fails when executed on the same version of a program. Flakiness hinders CI cycles and prevents automatic builds due to false signals, resulting in undesirable delays. Furthermore, surveys [6]–[8] show that flakiness affects developers' productivity, as they spend a considerable time and effort investigating the nature and causes of flaky tests.

To alleviate this issue, researchers have proposed tools that help detect flaky tests. In particular, IDFlakies [9] and Shaker [10] detect flakiness in test suites by running tests in different setups. However, rerunning tests, especially for a large number of times, is resource-intensive and might not be a scalable solution. For this reason, researchers recently suggested alternative approaches to detect flaky tests based on features that do not require any test execution [4], [11]–[13]. Although promising, these approaches mainly focus on classifying tests as flaky or not without any additional explanation. Unfortunately, the absence of additional information prevents a proper comprehension of flaky failure causes. Hence, further investigation is required to understand the nature of flakiness and identify the culprit code elements that need to be fixed [7].

Another important line of research in the area regards automated approaches that aim at helping to locate the root causes and suggest potential flakiness fixes [14]–[16]. However, research on automatically fixing flakiness is still at an early stage: tools often focus on one category of flakiness and with few examples. For instance, iFixFlakies [17] and ODRepair [18] focus only on dealing with test order dependencies, which is one of the main causes of test flakiness. Flex [19] automatically fixes flakiness due to algorithmic randomness in machine learning algorithms.

We believe that both developers and researchers would benefit from additional information that could assist them in gaining a better understanding of flaky tests, once they have been detected. Therefore, we propose FlakyCat, a learning-

based flakiness categorization approach that identifies the key reason/category of the test failures.

One limitation of previous work, relying on supervised learning, regards the need for large volumes of available data. Unfortunately, debugged flaky test data is scarce, inhibiting the application of learning-based methods. To deal with this issue, we leverage the Few-Shot learning capabilities of Siamese networks, which we combine with the CodeBERT representations to learn flakiness categories from a limited set of data (flaky tests).

To evaluate FlakyCat, we gather a set of 451 flaky tests annotated with their category of flakiness issued from previous studies and projects that we mined from GitHub.

Our empirical evaluation aims at answering the following research questions:

- **RQ1:** How effective is FlakyCat compared to approaches based on other combinations of test representation and classifier?
  Findings: Our results show that FlakyCat is capable of predicting flakiness categories with an F1 score of 73%, outperforming classifiers based on traditional supervised machine learning.
- **RQ2:** How effective is FlakyCat at predicting each of the considered flakiness categories?
  Findings: FlakyCat classifies accurately flaky tests related to *Async waits*, *Test order dependency*, *Unordered collection*, and *Time*, with the best F1 score of 81% for the *Async waits* category. However, the approach shows difficulty in classifying concurrency-related flaky tests (an F1 score of 39%), since these cases are related to the interaction of threads and processes and are easily confused with Asynchronous waits.
- **RQ3:** How do statements of the test code influence the predictions of FlakyCat?
  Findings: We found that some statement types are specific to certain flakiness categories. This is the case for assert statements in *Unordered collections* and statements using date or time for the *Time* category. We also found that some flaky categories have similar statement types like the presence of thread usages in both *Async waits* and *Concurrency* categories.

In summary, our contributions can be summarized as follows:

- **Dataset** We collected 451 flaky tests alongside their categories.
- **Model** We present FlakyCat, a new approach using Few-Shot Learning and CodeBERT to classify flaky tests based on their flakiness category.
- **Interpretability** We introduce a novel technique to explain what information is learnt by models using CodeBERT as code representation.

To enable the reproducibility of our work, we make the dataset used to evaluate FlakyCat and the scripts publicly available in our replication package [1].

[1]https://github.com/Amal-AK/FLAKYCAT

The paper is organized as followed: Related works are presented in Section II. Section III presents the designed implementation of FlakyCat. Section IV introduces our interpretability technique. Section V describes how we collected our dataset and evaluated our study. Section VI presents the results of our study. We further discuss different use cases in Section VII. Finally, Section VIII discusses threats to the validity of this study.

II. Related Work

Recently, practitioners from the industry reported struggling with flakiness and highlighted the need to find solutions to the problem [3], [4], [20]–[23]. Consequently, researchers from academia started to draw their attention to the matter. Luo *et al*. presented the first empirical study to understand and categorize the root causes of flakiness, they analyzed 201 flaky tests and identified 10 root causes of flakiness, the top ones being Asynchronous waits, concurrency, and test order dependency. Using the same taxonomy defined by Luo [24], Eck *et al*. [25] classified 200 flaky tests and identified four new causes of flakiness. Over the years, several surveys were carried on to identify the sources, impacts and existing strategies to mitigate flakiness by interrogating developers and practitioners [6]–[8], [26]. Parry *et al*. presented the state of the art of academic research in another survey [27]. Researchers presented different tools and approaches to detect flaky tests in a more efficient way. Notably, DeFlaker [28], IDFlakies [9], Shaker [10] and NonDex [29] attempt to facilitate the detection of flaky tests compared to exhaustive reruns. Because the cost of running tests is viewed as expensive, researchers also sought to suggest static alternatives for the detection. Different approaches relying on machine learning were introduced. Pinto *et al*. [30] and following replication studies [31], [32] presented a vocabulary-based model using elements from the test code to classify tests as flaky or not. Others investigated the use of test smells [13] and code metrics [33] for predicting flaky tests. Trying to outperform the performances of existing approaches, others relied on a mix of static and dynamic features, like FlakeFlagger [34] or Flake16 [35]. Fixing flakiness is also an aspect that has recently been investigated. Shi *et al*. introduced iFixFlakies [17] to fix order-dependent flaky tests. At Google, Ziftci *et al*. suggested using coverage differences, between passing and failing executions of flaky tests to guide developers to understand the underlying problem. Coverage information is also used by FlakyLoc [16], which leverages spectrum-based fault localization to locate the root cause of flakiness in web apps. Logs are also frequently considered to be a useful source of information in understanding root causes of flakiness [15]. Closer to our work, Flakify [36] used CodeBERT [37] as a pre-trained language-based model for their predictor but their goal is to classify tests as flaky or not. To our knowledge, we are the first to focus on predicting the category of flakiness for each test. Few-Shot Learning is widely used in computer vision [38]. In software engineering though, fewer studies used this approach for their task. Notably, studies suggested using

this model for vulnerability detection [39] and code clone detection [40], but none were carried out for flakiness. About pre-trained language models, Wan *et al*. [41] investigated their ability to capture the syntax structure of source code and report that they are efficient for code processing tasks.

III. FLAKYCAT

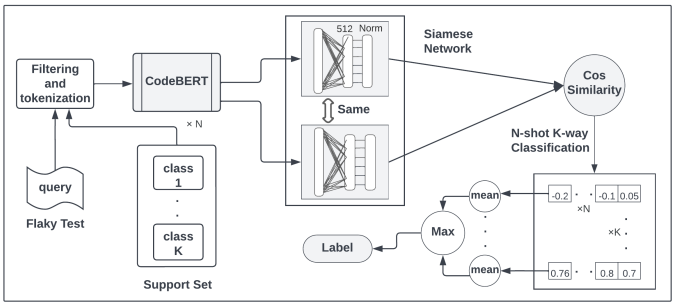

Fig. 1. An overview of FlakyCat, which combines the use of the pre-trained model CodeBERT, and Few Shot Learning based on the Siamese network.

In this section, we present the design and implementation of our approach. Figure 1 presents an overview of the main steps of FlakyCat, code transformation and classification.

A. Step 1: Flaky test transformation

1) Scope: We rely on the test code to assign flaky tests to different categories. Previous studies showed that flakiness finds its root causes in the test in more than 70% of the cases [24], [42]. Hence, focusing on the test code allows us to capture the nature of flakiness while minimizing the overall cost of FlakyCat. Indeed, considering the code under test would require running the tests and collecting the coverage, which entails additional requirements and costs.

2) Flaky test vectorization: In order to perform a source code classification task, we first need to transform the code into a suitable representation that will be fed to the classification model. Among previous studies predicting flaky tests statically, two main approaches were used to transform code into vectors: using test smells [13], [34] and using code vocabulary [12], [31], [32]. Both approaches seem promising, as different studies report high-performance models. As their encoding enables flaky test prediction, we believe they could also be used for flakiness category prediction, and we compare them with our approach.

Recently, code embeddings from pre-trained language models were also considered for source code representation [36], [43]. Pre-trained language models allow the encoding of code semantics and are intended for general-purpose tasks such as code completion, code search, and code summarization. Considering these benefits, we use the pre-trained language model CodeBERT [44] to generate source code embeddings. CodeBERT can learn the syntax and semantics of the code and doesn't require any predefined features [41]. Considering this aspect, we decide to rely on the CodeBERT test representation.

CodeBERT has been developed with a multi-layer transformer architecture [45] and trained on over six million pieces of code involving six programming languages (Java, Python, JavaScript, PHP, Ruby, and Go).

To get the code representation using CodeBERT model, we first filter out extra spaces such as line breaks and tabs from the source code. In our case, we use each test method's source code as individual sequences. We then tokenise sequences by converting each token into IDs. Each sequence is passed to the CodeBERT model, which returns a vector representation. Figure 2 illustrates this process.

Next, we explain the inputs and outputs of CodeBERT.

a) Inputs: CodeBERT is able to process both source code and natural language, *e.g.*, comments and documentation. In our case, we did not exploit the possibility of using comments as the input length of CodeBERT is limited. Furthermore, comments can add noise since they represent unstructured text, possibly written by different developers, so we decided to solely rely on the code semantics. Hence, the given input to CodeBERT only considers code tokens, surrounded by two special tokens for boundaries. This is represented as follows:

$$[CLS], c1, c2, ..., cm, [SEP].$$

Where *Ci* is a sequence of code tokens, the special token [SEP] indicates the end of the sequence, and [CLS] is a special token placed in the beginning, whose final representation is considered as the representation of the whole sequence which we use for classification.

b) Outputs: CodeBERT output includes two representations. The first one is the context matrix where each token is represented by a vector, and the second one is the CLS representation, having a size of 768, which is an aggregation of the context matrix and represents the whole sequence. For the purpose of FlakyCat, we are interested in the CLS vector that represents the complete test code.

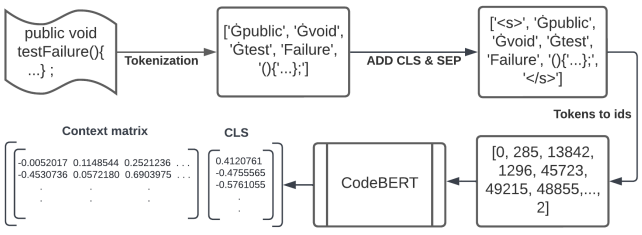

Fig. 2. The process of converting the source code of each test case to a vector using CodeBERT, going through tokenization, then converting to IDs and applying the CodeBERT model to get the representation (CLS vector). Ġ represent spaces, $< s >$ used for CLS, and $< /s >$ for SEP.

B. Step 2: Flaky test categorization

1) Classification process: Unlike traditional machine learning classifiers that attempt to learn how to match an input x to a probability y by training the model in a large training dataset and then generalizing to unseen examples, Few-Shot Learning (FSL) classifiers learn what makes the elements similar or

belonging to the same class from only a few data. Facing the scarcity of data on flaky tests, selecting an FSL classifier seems then to be a promising choice.

In FSL, we call the item we want to classify a *query*, and the *support set* is a small set of data containing few examples for each class used to help the model to make classifications based on similarity as shown in Figure 1. To classify flaky tests according to their flakiness category, we compute the similarity between the query and all examples of each flakiness category in our Support Set and assign the label having the maximum similarity with the query. This classification is obviously performed in a space where all elements of the same class are similar or close to each other. This is achieved by a model called *Siamese network*. Its task is to transform the data and project it into a space where all the elements of a same class are close to each other, and then to classify the elements by computing their similarity.

The Siamese network has knowledge of the similarity of elements of the same class. It processes two vectors in input and applies transformations that allow minimizing the distance between the two vectors if they share similar characteristics. Figure 3 shows an example of the visualization of flaky test vectors before and after the Siamese network is applied. Since CodeBERT has no knowledge of the characteristics of flaky tests and only generates a general representation of the source code, the vectors produced are all similar. However, the Siamese networks learn which characteristics in these vectors are shared by tests of the same class, and thus allow to project vectors into a space that groups tests of the same flakiness category. After this step, it becomes possible to classify them using a similarity computation.

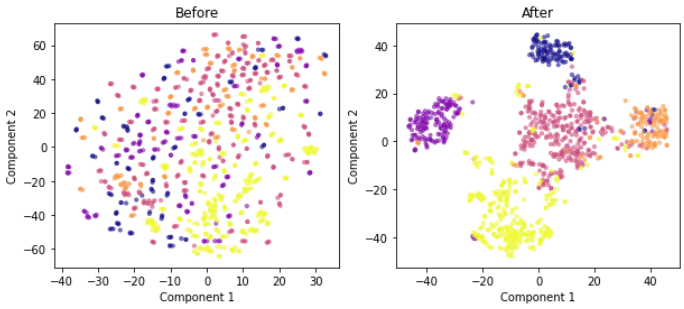

Fig. 3. Visualization of our data before and after training of the Siamese network with the triplet loss, which brings together the elements of the same class.

2) Model training: Siamese networks have two identical sub-networks, each sub-network processes the input vector and performs transformations. Both sub-networks are trained by calculating the similarity between the two inputs and using the similarity difference as a loss function. Accordingly, the weights are adjusted to have a high similarity if the inputs belong to the same class. For the architecture of the sub-networks, we used a dense layer of 512 neurons and a normalization layer as shown in Figure 1. We also performed a linear transformation to keep relations learnt by CodeBERT using the attention mechanism introduced in the transformer architecture [46]. This model is trained using a Triplet Loss function, based on the calculation of similarity difference.

Let the Anchor A be the reference input (it can be any input), the positive example P is an input that has the same class as the Anchor, the negative example N is an input that has a different class than the Anchor, $s()$ is the cosine similarity function, and m is a fixed margin. The idea behind the Triplet Loss function is that we maximize the similarity between A and P, and minimize the similarity between A and N, so ideally $s(A, P)$ is large and $s(A, N)$ is small. The formula for this loss function is:

$$Loss = max(s(A, N) - s(A, P) + m, 0)$$

m is an additional margin as we do not want $s(A, P)$ to be very close to $s(A, N)$, which would lead to a zero loss.

To train the Siamese network with the triplet loss, we give as input batches of pairs with the same classes, and any other pair of a different class can be used as a negative example. We select the closest negative example to the anchor, such as $s(A, N) \simeq s(A, P)$, which generates the largest loss and constitutes a challenge for model learning.

IV. Interpretability Technique

Model interpretability refers to one's ability to interpret the decisions, recommendations, or in our case the predictions, of a model. Interpretability is a crucial step to increase trust in using a machine learning model. Indeed, it allows model creators to investigate potential biases in the learning processes and better assess the overall performance of their models. On top of that, providing developers with information about how the model came to its prediction can enhance the model adoption [47].

Flakiness prediction approaches often relied on Information Gain to explain what features in the model appeared to be the most useful [13], [30], [34]. In the case of tree-based models, the reported information gain is given by the Gini importance (also known as Mean Decrease in Impurity) [48]. Parry *et al.* [35] used SHapley Additive explanations (SHAP), which is another popular technique for model interpretability [49].

As FlakyCat uses the CodeBERT representation of tests as input, using the previously mentioned techniques would not give understandable features. To our knowledge, there are no existing techniques used for CodeBERT-based model interpretability. Thus, we introduce a novel approach to better understand the decisions of CodeBERT-based models. Following the main motivation of helping developers better understand flaky tests once detected, our goal with this interpretability technique is to arm FlakyCat users with a more fine-grained explanation for the model's decision.

Our technique is inspired by delta debugging algorithms. Delta debugging is used to minimize failure-inducing inputs to a smaller size that still induces the same failure [50]. In our case, we are interested in the particular code statements linked with the most influential information for the model's decision.

To identify them, we proceed with the following: We classify all the original test cases and save their similarity scores. We create new versions of each test. Each version is a copy of the original test minus one statement that was removed. Next, we feed the new versions to FlakyCat. Among all new versions for one test, we keep the one for which the similarity score endured the biggest drop compared to the original prediction score. We consider the statement removed in this version as the most influential one.

V. Evaluation

In this section, we explain our evaluation setting for FlakyCat. First, we describe our data curation process, then, we present our approach for answering each of the three research questions.

A. Data curation

1) Collection: For our study, we had to collect a set of flaky tests containing their source code and their flakiness category. We focused our collection efforts on one programming language, as training a classifier using code and tokens from different programming languages is more challenging. For the language choice, we opted for Java, which is the most common language in previous flakiness studies (and thus datasets). To increase the amount of data used in this study, we also collected a new set of flaky tests mined from GitHub that we classified manually.

TABLE I
Data filtering performed on the different datasets used in this study. Collected represents the new dataset we retrieved.

Filters	Datasets				
	[24]	[51]	[52]	[17]	Collected
Inspected commits	201	170	40	101	270
Commit not found	12	12	4	3	3
Duplicated commit	0	2	0	0	3
Open commit	0	0	0	33	0
Flaky test not found	45	21	13	0	42
Configuration problems	3	8	0	0	0
Not Java	15	5	0	0	8
Category hard to classify	40	57	4	0	22
Considered commits	86	65	19	65	192
Total of extracted tests	109	65	20	65	192

a) Existing datasets: There is no large public dataset of flaky tests labelled according to their category of flakiness. Most of the existing studies, such as FlakeFlagger [34] and DeFlaker [53], are limited to list detected flaky tests which are later used for binary classification. Regarding the data classified into flakiness categories defined by Luo *et al*. [24] and Eck *et al*. [25], there is only limited data available in previous empirical studies about flakiness. We retrieved tests from the empirical study of flaky tests across programming languages of Costa *et al*. [51] and from a recent study about pinpointing causes of flakiness by Habchi *et al*. [52]. We also retrieved the flaky tests from iFixFlakies [17] as *Test order dependency* is a flakiness category that received a large interest in the community [9], [18], [54], [55].

We gathered a total of 512 commits/pull requests from the existing datasets we could access, referenced in Table I.

b) New dataset: To expand existing datasets, we explore GitHub projects and search for flakiness-fixing commits for which developers explained the reason (*i.e.*, category) of flakiness.

In this search, we use flakiness-related keywords such as *Flaky* and *Intermit* in the commit messages. To ensure that the commit refers to a flakiness category, we further filter commits by specific keywords related to each category: *thread, concurrence, deadlock, race condition* for Concurrency, *time, hour, seconds, date format, local date* for Time, *port, server, network, http, socket* for Network and *rand* for Random. After the search, we rely on the developer's explanation in the commit message and on the provided fix to classify tests into the different flakiness categories listed in the literature. This collection allowed us to obtain 270 commits fixing flaky tests to be classified manually.

2) Filtering: The previous step allowed us to collect a total of 782 categorized commits/issues. In this step, we filter out commits and data that are not adequate for our study. We filter out commits hard to classify, duplicated ones, and those where flaky tests are not written in java. Costa et al. [51] classified issues, and Luo et al. [24] classified old SVN revisions. In some cases, the corresponding commit could no longer be found in the projects. Some data points were missing necessary attributes, such as the name of the flaky test. Particularly, in commits where the fix is in the production code or in a configuration file, and the test name of the involved flaky test is not indicated in the commit message, we were not able to identify the flaky test, so we filtered them out. The number of tests extracted for each dataset is shown in Table I. The *considered commits* row accounts for commits where all information needed was present *i.e.*, the test name, source and category of flakiness. Note that the number of considered commits and extracted tests vary in some cases as developers sometimes addressed more than one flaky test per commit. We obtained a total of 259 flaky tests after filtering the existing datasets. For the data we collected ourselves, we successfully extracted 192 test cases. To ensure the correctness of our manual classification and filtering, the first two authors of the paper performed a double-check on the newly collected dataset.

3) Processing: After filling in all the necessary attributes: the test case name, flakiness category, test file name, and project URL, we download the code files and extract test methods using the spoon library[2]. At this stage, all comments have been deleted from the source code to restrict CodeBERT to code statements.

4) Final dataset: The final dataset contains 451 flaky tests distributed over 13 flakiness categories. Table II illustrates this distribution.

The collected flaky tests are not distributed evenly across categories of flakiness. Just as shown in past empirical stud-

[2]https://github.com/INRIA/spoon

ies [24], [56], some categories, such as *Async waits*, are more prevalent than others. Our approach uses FSL to learn from limited datasets. Still, it requires a certain amount of examples to learn common patterns from each category. We decided to have at least 30 tests in a category to consider it. This number is commonly accepted by statisticians as a threshold to have representativeness [57], since learning from very few examples is not feasible. In our dataset, some flakiness categories contain no more than 5 flaky tests. We were not able to gather more data for those non-prevalent categories and thus decided to focus on five of the most common flakiness categories, highlighted in grey in the table: *Async waits*, *Test order dependency*, *Unordered collections*, *Concurrency*, and *Time*.

5) Data augmentation: Facing the challenge of learning from few data, we over-sampled our training set similarly to SMOTE [58] by applying elementary perturbations. In the same way, as we increase the imagery data by rotating and resizing, for the source code, we generate variants of our tests by mutating only the code elements that have no influence on flakiness. This includes variable names, constants such as strings, test method names, and by adding declarations of unused variables. In this way, the model will learn useful code elements instead of learning from variable names and strings. We used the Spoon library for the detection of these elements, and we replaced them with randomly generated significant words. As a result, the total number of tests after data augmentation is 964.

TABLE II
FINAL DATASET. THE HIGHLIGHTED ROWS ARE THE DATA USED TO TRAIN AND TEST THE MODEL. THE ORIGINAL DATA REFERS TO THE DATA WE COLLECTED, SHORT DATA ARE TESTS WITH LESS THAN 512 TOKENS, AND THE AUGMENTED DATA ARE THE DATA WE OBTAINED AFTER AUGMENTATION.

Class	**Data**		
	Original	***Short***	***Augmented***
Async waits	125	97	300
Test order dependency	103	100	284
Unordered collections	51	48	146
Concurrency	48	40	124
Time	42	38	110
Network	31	25	/
Randomness	17	14	/
Test case timeout	14	9	/
Resource leak	10	7	/
Platform dependency	2	2	/
Too restrictive range	3	2	/
I/O	2	2	/
Floating point operations	3	1	/
TOTAL	451	385	964

B. Experimental design

1) Baseline:

To the best of our knowledge, we are the first to introduce an automatic classification of flaky tests according to their category. However, to get a better appreciation of the performance of the solution we propose in this paper, we seek to compare FlakyCat with test representations commonly used by flaky test detection approaches. Our intuition is that test representations giving good performance in binary classification (*i.e.*, detecting flaky tests and non-flaky tests) have a good chance to be helpful for the classification of tests according to their category of flakiness. Thus, we use the following representations for our multi-classification task: the vocabulary-based approach [12] which is a keyword-based approach, and the smell-based approach [13] which exploits the correlation between test smells and test flakiness. Our overall motivation is to determine whether it is possible to make this classification based on limited data and to know which combination of classifier and code representation delivers the best results.

For the classification based on test smells, we use the 21 smells detected by tsDetect [59], to generate vectors indicating the presence of each smell detected by the tool, in the same way as in the study of Camara *et al*. [13]. As for the vocabulary-based classification, we use token occurrence vectors, as in the article by Pinto *et al*. [12]. We tokenize the code and apply standard pre-processing like stemming, then calculate occurrences of each token.

In addition to various test representations, we compare our FSL-based approach with traditional classifiers from the Scikit-learn library [60] used by previous studies on flakiness prediction [12], [13], [32]: Random Forest (RF), Support Vector Machine (SVM), Decision Tree (DT) and K-Nearest Neighbour (KNN).

To validate each model, we split our data into 75% for training and 25% for final validation. We use a 10-fold stratified cross-validation on the training data to select the best model parameters and use those parameters to evaluate the model on the unseen hold-out set.

As the augmented samples in our dataset are variants of the original ones, it was important to keep them in the same sets, to ensure that no similar data pairs are included in both the training and test sets. For the support set used for classification, we select the most centred examples to represent each class.

FlakyCat relies on a Siamese network. It is trained with combinations of data by indicating whether these data are similar or not so that the model can learn what makes them similar. Since we train with combined data, the balancing of data is not required, because it is automatically over-sampled.

2) Parameters:

We tuned FlakyCat's parameters on the training set using the Random Search method [61] and a 10-fold cross-validation, by testing random combinations of the most important parameters that have a direct impact on the model performance, which include the similarity margin used in the triplet loss function, the learning rate, the number of warm-up steps, and the support set size.

Figure 4 shows the resulting weighted F1 score for each tested parameter combination using a 10-folds cross-validation. A high learning rate and a number of warm-up steps have a negative impact on the performances, while other parameters have a lower influence. Following these results, for the final validation on the hold-out set, we use the best parameter combination identified in the Figure 4: a similarity

margin of 0.30, a learning rate of 0.001, a number of warm-up steps of 400, and a support set with 10 examples from each category.

For baseline classifiers, we keep the standard values used by previous works. We varied the number of trees in the Random Forest classifier, we tested values from 100 to 1000 with a step of 100. We observed that this does not make much difference regarding the F1 score ($\leq$ 3%), and we identified 1000 as the number giving the best results.

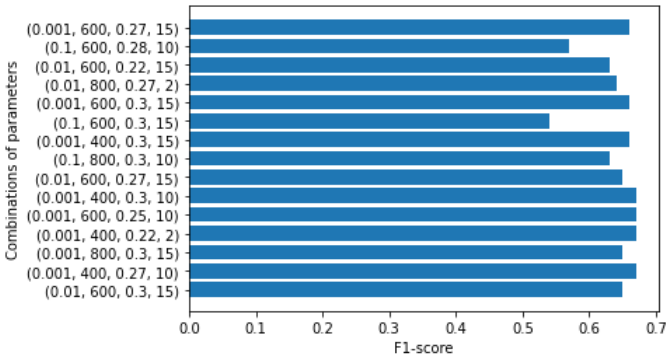

Fig. 4. F1 score for different values of parameter combinations using Random Search and a 10-Folds cross-validation. The combinations on the Y axis have the form : (learning rate, number of warm-up steps, similarity margin, support set size).

3) Evaluation metrics:
We use the standard evaluation metrics to compare classifiers, including precision, recall, Matthews correlation coefficient (MCC), F1 score, and Area under the ROC curve (AUC). These metrics have been used to evaluate the performance of classifiers, including binary classification of flaky tests [12], [13], [36]. Since our dataset is unbalanced, weighted metrics are more suitable for our evaluation.

4) Research questions:

a) ***RQ1: How effective is FlakyCat compared to approaches based on other combinations of test representation and classifier?****:* This question aims to evaluate FlakyCat and compare it to other test representation techniques, *i.e.*, vocabulary and test-smell-based and traditional classifiers, *i.e.*, SVM, KNN, decision tree and random forest.

b) ***RQ2: How effective is FlakyCat in predicting each of the considered flakiness categories?****:* This question evaluates FlakyCat's ability to classify the different categories of flakiness. To perform this, we split the dataset into five sets following the categories: *Async waits*, *Test order dependency*, *Unordered collections*, *Concurrency*, and *Time*. Then, we use the same settings as for RQ1 to tune the Siamese network, train it, and evaluate it for each category.

c) ***RQ3: How do statements of the test code influence the predictions of FlakyCat?****:* We applied the technique we introduced in Section IV for CodeBERT-based model interpretability to FlakyCat. We classified all original short data (323 tests). For 16 tests, the score doesn't decrease by deleting one statement, and thus we collected 307 statements of interest, important for FlakyCat's decision-making. To better understand what information emerges, we proceeded with the following analysis. First, we regroup statements by category of flakiness (according to the flaky test they belong to). Then, we want to share information on what type of statements FlakyCat found useful. To do so, we look through the list of statements and attempt to identify recurring code statements and categorize them. The process of identifying statement types is exploratory and inspired by qualitative research. Two of the authors of this paper went through the list of statements and identified nine recurring types of statements:

- **Control flow:** Includes decision-making statements, looping statements, branching statements, Exception handling statements.
- **Asserts:** All types of assertions in tests.
- **Threads:** statements related to threads and runnables.
- **Constants:** Constant values such as strings, numbers and boolean values independent of variables, and final variables.
- **Waits:** All explicit wait statements.
- **Usage of date/time:** Statements that perform operations on time values, dates.
- **Network:** Statements related to data exchange in a local or external network between two endpoints, and session management.
- **I/O:** Statements related to input/output, database and file access.
- **Global variables:** Includes the use of global variables.

With this question, we investigate the prevalence of these statement types in each flakiness category.

VI. Results

A. RQ1: How effective is FlakyCat compared to approaches based on other combination of test representation and classifier?

Following the outlined experimental design, we trained and tested FlakyCat and the four traditional classifiers, using the three source code representations, the vectors obtained from CodeBERT, the vectors based on vocabulary, and the ones based on test smells. The obtained results are presented in Table III. The results show that FlakyCat achieves the best performance for all evaluation metrics. It obtained an average weighted F1 score of 73% and a precision of 74%. We get an MCC of 0.65 (bounds for this metric are between -1 and 1), being close to 1 means a perfect classification. Finally, the AUC of 0.83 shows that the model is able to distinguish flaky tests from different classes.

a) Representation effect: Regarding the three code representations, CodeBERT achieves the best performance for RF, KNN, and FSL, with an F1 score between 0.51 and 0.73 for the three classifiers. When using the vocabulary-based vectors, SVM and DT perform better than using CodeBERT. With this representation, all classifiers do not exceed an F1 score of 0.67. The representation based on test smells yields lower results, with the best F1 score being 0.29. The CodeBERT representation seems then promising to use when learning to classify flaky tests according to their categories.

TABLE III
COMPARING PERFORMANCES OF FLAKYCAT (CODEBERT AND FEW-SHOT LEARNING) WITH TRADITIONAL MACHINE LEARNING CLASSIFIERS

Model	Smells-based					Vocabulary-based					CodeBERT-based				
	Precision	*Recall*	*MCC*	*F1*	*AUC*	*Precision*	*Recall*	*MCC*	*F1*	*AUC*	*Precision*	*Recall*	*MCC*	*F1*	*AUC*
SVM	0.11	0.34	0.00	0.17	0.50	0.61	0.52	0.37	0.45	0.66	0.27	0.43	0.22	0.33	0.60
KNN	0.24	0.37	0.11	0.29	0.55	0.44	0.48	0.31	0.45	0.65	0.56	0.53	0.37	0.51	0.68
DT	0.31	0.33	0.10	0.23	0.53	0.53	0.53	0.39	0.52	0.69	0.49	0.50	0.34	0.49	0.67
RF	0.32	0.34	0.12	0.24	0.54	0.72	0.61	0.49	0.56	0.72	0.68	0.66	0.55	0.62	0.76
FSL	0.13	0.18	-0.01	0.13	0.50	0.69	0.68	0.58	0.67	0.79	**0.74**	**0.73**	**0.65**	**0.73**	**0.83**

b) Classifier effect: Regarding the choice of classifier, we find that the FSL classifier based on similarity achieves the best performance using the representations based on CodeBERT and vocabulary. Among traditional classifiers, Random Forest obtains the best results, as reported in previous flaky test classification studies [30], [31]. Classifiers relying on the smell-based representation have more difficulty to classify flaky tests. Using this code representation, the KNN classifier achieved the best F1 score: 0.29. Two categories had a positive impact to achieve this score: *Async wait*, and *Test order dependency*. This can be explained by the presence of test smells strongly related to these two categories, including Sleepy test and Resource optimism. Other flakiness categories seem to be more challenging to predict using existing test smells.

c) Random-guessing comparison: In the previous paragraph, we compared different models and different code representations and saw that FlakyCat gave the best results. Because all the existing approaches were designed to detect flaky tests from non-flaky tests, they might not be suitable for the specific task of classifying flaky tests according to their categories. As no other category-based classification technique exist so far, we show the performance of a random guesser as another baseline. We consider two random guessing approaches, the first one where we randomly affect a class to each flaky test and the second one where we weigh the random affectation according to the prevalence of flaky tests in each category. Both approaches are considered as the dataset balance might be different from the one found in various projects. Results are listed in Table IV. F1 scores for Random and Weighted Random are respectively of 0.21 and 0.25. With an F1 score of 0.73, we see that FlakyCat performs better than the two considered random-guessing approaches.

RQ1 Overall, our results show that automatic classification of flaky test categories with a limited amount of data is a challenging but feasible and promising task. The representation based on CodeBERT gives better results compared to the ones based on test smells and vocabulary. We also found that Few Shot Learning performs better than traditional machine learning classifiers.

TABLE IV
PERFORMANCE OF RANDOM GUESSING APPROACH

Method	*Precision*	*Recall*	*MCC*	*F1*	*AUC*
Random	0.25	0.20	-0.01	0.21	0.50
Weighted Random	0.25	0.26	0.02	0.25	0.51

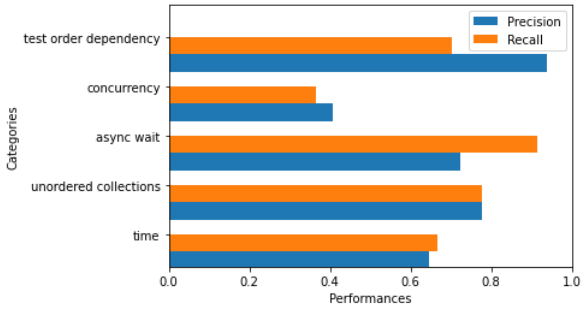

Fig. 5. Precision and Recall per flakiness category using FlakyCat

B. RQ2: How effective is FlakyCat in predicting each of the considered flakiness categories?

Figure 5 shows performances achieved by FlakyCat for each of the five flakiness categories. Results show that the category *Async wait* is the easiest for the model to classify, with an F1 score of 0.81. The category *Test order dependency*, *Unordered collections* and *Time* respectively have an F1 score of 0.80, 0.78 and 0.66. *Concurrency* performances are lower with an F1 score of 0.39. We suspect that concurrency issues happen in many cases in the code under test. As FlakyCat only relies on the test source code, this would indeed explain why performances are lower in this case. Another supposition is that concurrency issues and asynchronous waits are sometimes closely related. We discuss an example of this in Section VII-A.

RQ2 While the four flakiness categories *Async waits*, *Test order dependency*, *Unordered collections*, and *Time* show good ability to be detected automatically, *Concurrency* remains difficult to detect by relying only on the test case code.

TABLE V
PREVALENCE OF DIFFERENT TYPES OF STATEMENTS IN EACH FLAKINESS CATEGORY FOR TRUE POSITIVE PREDICTIONS

	#Statements	Control flow	Constants	Asserts	Threads	Waits	Network	Global variables	Usage of Date/time	I/O
Async Waits	80	16,25%	55%	20%	20%	27,5%	25%	11,25%	3,75%	6,25%
Concurrency	34	23,53%	47,06%	17,65%	29,41%	14,70%	14,70%	5,88%	17,65%	2,94%
Test order dependency	69	8,69%	60,87%	13,04%	0%	4,35%	8,69%	2,90%	7,25%	47,82%
Time	32	18,75%	56,25%	50%	0%	0%	0%	9,375%	62,5%	6,25%
Unordered collections	42	4,76%	66,67%	38,09%	0%	0%	2,38%	4,76%	0%	4,76%

C. RQ3: How do statements of the test code influence the predictions of FlakyCat?

Table V reports the prevalence (%) of the different types of statements among all influential statements per flakiness category, *e.g.*, 100% Asserts in the *Time* category would mean that all influential statements for the *Time* category contain assert statements.

Compared to other flakiness categories, the percentage of assertions in the influential statements of *Time* and *Unordered collections* is high, 50% and 38.09% respectively. Based on our analysis, this includes in particular assertions that perform exact comparisons, such as `assertEquals()`, between constant values and collection items, or dates for example. 29,41% of influential statements in the *Concurrency* category include thread manipulation, and 20% for the *Async Waits* category, while the rest of the categories have none. Statements containing explicit waits represent respectively 27,5% and 14,7% for *Async Waits* and *Concurrency* categories, but below 5% for *Test order dependency* and zero for the others. Statements containing date or time values are most common in the *Time* category with 62,5%. We note that they appear as well in a small proportion, 17,65%, for *Concurrency*. Statements from the I/O calls group are mainly found in the *Test order dependency*. For *Control flow*, *Constants*, and *Global variables* statements are almost evenly distributed. We include a spreadsheet containing all statements analyzed in our replication package.

RQ3 The interpretability technique we presented enable us to find which statements impact FlakyCat's decision. We also find hints that specific flakiness categories have distinct statement types (*e.g.*, Usage of Date/time for the *Time* category) while some others have similar prevalence (*e.g.*, Threads for *Async Waits* and *Concurrency* categories). By highlighting these statements, our interpretability technique may provide information to developers to better understand flaky tests, their categories and their causes.

VII. DISCUSSION

A. Reasoning about the statements influencing FlakyCat and the usage of flakiness categories

Listing 1 gives an example of a flaky test taken from the Neo4J project[3] found during the data collection part. As explained in the commit message, the flakiness was caused by a race condition and thus, we affected it to the Concurrency category. FlakyCat classified this test as Async wait. The interpretability technique that we introduced in Section IV reveals that the statement on line 6 is the most influential for the model's decision. It contains the `await()` function, and this is likely the reason why the flaky test was categorized as Async wait. Furthermore, similarity score for the Concurrency category is high, and it comes as FlakyCat's second guess.

When looking at the test, we understand that an asynchronous wait was performed to wait for a thread. We also found similar examples concerning other categories, such as waits relying on network resources. First, we argue that our interpretability technique can help to understand the cause of flakiness, even when FlakyCat apparently mislabelled the test. Secondly, we advance that flakiness categories as commonly defined in research studies [7], [24] can overlap, *i.e.*, a flaky test can belong to several categories. The application of machine learning to determine the causes of flakiness is promising and should receive attention. It would also benefit from a more precise, orthogonal classification of flakiness categories.

```
1  @Test
2  public void shouldPickANewServer[...]() throws
       Throwable {
3  [...]
4      Thread thread = new Thread( () -> {
5      try {
6      startTheLeaderSwitching.await();
7      CoreClusterMember theLeader = cluster.
8          awaitLeader();
9      switchLeader( theLeader );
10     } catch ( TimeoutException |
       InterruptedException e ) {
11         // ignore
12     }});
13 [...]
14 }
```

Listing 1. A flaky test belonging to two categories

B. The effect of considering an additional category

Our results showed that flakiness categories can be classified automatically. We carried out our main experiments with five categories of flakiness for which we had a reasonable number of tests. Still, we believe that one interesting aspect of our study is to understand the impact of adding other categories to FlakyCat. For this, we investigate the performance of FlakyCat for each category (similarly to RQ2), but we add to our set the *Network* category, which is the next category with the most

[3]https://github.com/neo4j/neo4j/commit/c77e579b40b02087

samples in our dataset (25 tests). F1 scores and the accuracy obtained for each category are presented in Figure 6.

Compared to the results previously reported in Table III, we observe that the performances of each category are slightly impacted. The *Async waits* category is the most impacted one. Indeed, after adding the *Network* category, we get an overall F1 score of 0.68. The added category gets the worst results. This performance drop can be explained by multiple factors. First, having more categories to differentiate makes it more challenging for FlakyCat to distinguish between them. Secondly, the overall F1 score is strongly affected by the poor performance observed in the new category. The performance for the Network category can be a result of the too low number of examples in this category (25). Despite using FSL, the model still requires enough data points in each category. While collecting data, we noticed that flaky tests caused by Network were not common. These findings align with the ones about the prevalence of the different categories reported in previous empirical studies [7], [24]. In addition, flaky tests related to *Network* issues could also be considered as Asynchronous waits in many cases, as previously explained.

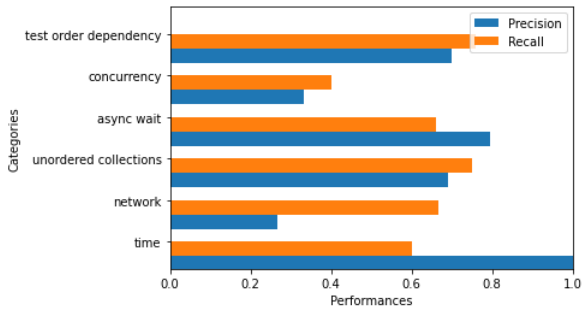

Fig. 6. Precision and Recall per flakiness category when adding the category "Network"

VIII. Threats to validity

a) Internal validity: One threat to the internal validity is related to the dataset we used in our study. Flaky tests were gathered from different sources, as explained in Section V-A. It is possible that flaky tests were assigned to the wrong label, which would impact the training and evaluation of our model. Certifying the category based on the test source code is complex and can as well be subjective. To ensure the quality of the data, the first two authors reviewed the collected flaky tests and confirmed their belonging to the assigned category.

Similarly, the identification of statement types in RQ3 required a manual analysis of the most influential statements. Hence, the identified types can be subjective and the assignment of statements is prone to human errors. To mitigate this risk, we kept the statement types factual, *e.g.*, control flow and asserts. This allows us to avoid assignment ambiguities and intersections between the different statement types.

b) External validity: The first threat to external validity is the generalizability of our approach. In this study, we train a model to recognize flaky tests from four of the most prevalent categories, but we are not sure of the performances in other categories. We discussed the addition of two categories (Network and Randomness), and retrieved that the number of examples is one of the influencing factors.

c) Construct validity: One potential threat to construct validity regards the metrics used for the evaluation study. To alleviate this threat, we report MCC, F1 score, and AUC metrics in addition to the commonly-used precision and recall. As our data is not evenly distributed across the different categories, we report the weighted F1 score.

IX. Conclusion

Test flakiness is considered as a major issue in software testing as it disrupts CI pipelines and breaks trust in regression testing. Detecting flaky tests is resourceful, as it can require many reruns to reproduce failures. To facilitate the detection, more and more studies suggest static and dynamic approaches to predict if a test is flaky or not. However, detecting flaky tests constitutes only a part of the challenge, since it remains difficult for developers to understand the root causes of flakiness. Such understanding is vital for addressing the problem, *i.e.*, fixing the cause of flakiness. At the same time, researchers would gain more insights based on this information. So far, only a few automated fixing approaches were suggested and these are focusing on one category of flakiness. Knowing the category of flakiness for a given flaky test is thus a piece of key information.

With our work, we propose a new approach to this problem that aims at classifying previously identified flaky tests into their corresponding category. We propose FlakyCat, a Siamese network-based multi-class classifier that relies on CodeBERT's code representation. FlakyCat addresses the problem of data scarcity in the field of flakiness by leveraging the Few-Shot Learning capabilities of Siamese networks to allow the learning of flakiness categories from small sets of flaky tests. As part of our evaluation of FlakyCat, we collect and make available a dataset of 451 flaky tests with information about their flakiness categories.

Our empirical evaluation shows that FlakyCat performs the best compared to other code representations and traditional classification models used by previous flakiness prediction studies. In particular, we reach an F1 score of 73%. We also analyzed the performances with respect to each category of flakiness, showing that flaky tests belonging to *Async waits*, *Test order dependency*, *Unordered collections*, and *Time* are the easiest to classify, whereas flaky tests from the *Concurrency* category are more challenging. Finally, we present a new technique to explain CodeBERT-based machine learning models. This technique helps in explaining what code elements are learnt by models and give more information to developers who wish to understand flakiness's root causes.

Acknowledgment

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes.

REFERENCES

[1] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[2] M. Rehkopf, "What is continuous integration — atlassian," https://www.atlassian.com/continuous-delivery/continuous-integration, (Accessed on 01/12/2021).

[3] J. Micco, "The State of Continuous Integration Testing Google," 2017.

[4] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*. IEEE / ACM, 2019, pp. 101–110.

[5] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.

[6] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," *International Conference on Software Testing (ICST)*, 2022.

[7] M. Eck, M. Castelluccio, F. Palomba, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," *arXiv*, pp. 830–840, 2019.

[8] M. Gruber and G. Fraser, "A survey on how test flakiness affects developers and what support they need to address it," in *Proceedings of the 15th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '22, 2022.

[9] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322, 2019.

[10] D. Silva, L. Teixeira, and M. D'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 301–311, 2020.

[11] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a Bayesian Network Model for Predicting Flaky Automated Tests," *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 100–107, 2018.

[12] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 492–502.

[13] B. Camara, M. Silva, A. Endo, and S. Vergilio, "On the use of test smells for prediction of flaky tests," in *Brazilian Symposium on Systematic and Automated Software Testing*, 2021, pp. 46–54.

[14] C. Ziftci and D. Cavalcanti, "De-flake your tests : Automatically locating root causes of flaky tests in code at google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 736–745.

[15] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. Beijing, China: ACM Press, 2019, pp. 101–111.

[16] J. Morán, C. Augusto, A. Bertolino, C. de la Riva, and J. Tuya, "Flakyloc: Flakiness localization for reliable test suites in web applications," *J. Web Eng.*, vol. 19, no. 2, pp. 267–296, 2020. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.1927

[17] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies : A Framework for Automatically Fixing Order-Dependent Flaky Tests," in *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations ofSoftware Engineering (ESEC/FSE '19)*, 2019.

[18] C. Li, C. Zhu, W. Wang, and A. Shi, "Repairing order-dependent flaky tests via test generation," in *Proceedings of the 44th International Conference on Software Engineering - ICSE '22*. ICSE, 2022.

[19] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 211–224, 2020.

[20] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? automatic cause analysis for test alarms in system and integration testing," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2017.71

[21] M. contributors, "Test verification - mozilla — mdn," https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification, March 2019, (Accessed on 01/12/2021).

[22] M. Harman and P. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, sep 2018, pp. 1–23. [Online]. Available: https://ieeexplore.ieee.org/document/8530713/

[23] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests : Spotify engineering," https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/, November 2019, (Accessed on 01/12/2021).

[24] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, nov 2014, pp. 643–653.

[25] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi-org.sndl1.arn.dz/10.1145/3338906.3338945

[26] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Surveying the developer experience of flaky tests," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[27] O. Parry, "A Survey of Flaky Tests," *ACM transactions on software engineering and methodology*, vol. 31, no. 1, 2021.

[28] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. New York, New York, USA: ACM Press, 2018, pp. 433–444. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180155.3180164

[29] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "Nondex: A tool for detecting and debugging wrong assumptions on java api specifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 993–997.

[30] G. Pinto, B. Miranda, S. Dissanayake, M. D'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 492–502, 2020.

[31] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests," *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021.

[32] B. Camara, M. Silva, A. T. Endo, and S. Vergilio, "What is the vocabulary of flaky tests? an extended replication," in *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 444–454. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICPC52881.2021.00052

[33] V. Pontillo, F. Palomba, and F. Ferrucci, "Toward static test flakiness prediction: A feasibility study," in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, 2021, pp. 19–24.

[34] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584.

[35] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Evaluating features for machine learning detection of order-and non-order-dependent flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 93–104.

[36] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering*, pp. 1–17, 2022.

[37] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[38] X. Sun, B. Wang, Z. Wang, H. Li, H. Li, and K. Fu, "Research progress on few-shot learning for remote sensing image interpretation," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 14, pp. 2387–2402, 2021.

[39] M. Khajezade, F. H. Fard, and M. S. Shehata, "Evaluating few shot and contrastive learning methods for code clone detection," *arXiv preprint arXiv:2204.07501*, 2022.

[40] Y. He, W. Wang, H. Sun, and Y. Zhang, "Vul-mirror: a few-shot learning method for discovering vulnerable code clone," *EAI Endorsed Transactions on Security and Safety*, vol. 7, no. 23, p. e4, 2020.

[41] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?–a structural analysis of pre-trained language models for source code," *arXiv preprint arXiv:2202.06840*, 2022.

[42] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[43] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[44] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[46] ——, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[47] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.

[48] "Feature importances with a forest of trees — scikit-learn 1.1.1 documentation," https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html, (Accessed on 06/24/2022).

[49] S. Lundberg, "Shap documentation," https://shap.readthedocs.io/, 2018, (Accessed on 06/23/2022).

[50] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[51] K. Costa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, pp. 1–14, 2022.

[52] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Le Traon, "What made this test flake? pinpointing classes responsible for test flakiness," *arXiv e-prints*, pp. arXiv–2207, 2022.

[53] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 433–444.

[54] C. Li and A. Shi, "Evolution-aware detection of order-dependent flaky tests," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 114–125.

[55] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Evaluating features for machine learning detection of order-and non-order-dependent flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 93–104.

[56] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An Empirical Study of Flaky Tests in Python," *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021*, pp. 148–158, 2021.

[57] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities," *Educational and psychological measurement*, vol. 30, no. 3, pp. 607–610, 1970.

[58] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[59] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[61] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.