RESEARCH-ARTICLE

Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures

OWAIN PARRY, The University of Sheffield, Sheffield, South Yorkshire, U.K.

GREGORY M KAPFHAMMER, Allegheny College, Meadville, PA, United States

MICHAEL HILTON, Carnegie Mellon University, Pittsburgh, PA, United States

PHIL MCMINN, The University of Sheffield, Sheffield, South Yorkshire, U.K.

Published: 17 June 2025

EASE '25: Evaluation and Assessment in Software Engineering
June 17 - 20, 2025
Istanbul, Turkiye

EASE '25: Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (June 2025)
https://doi.org/10.1145/3756681.3756945
ISBN: 9798400713859

Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures

Owain Parry
University of Sheffield
Sheffield, United Kingdom
o.b.parry@sheffield.ac.uk

Gregory Kapfhammer
Allegheny College
Meadville, USA
gkapfham@allegheny.edu

Michael Hilton
Carnegie Mellon University
Pittsburgh, USA
mhilton@cmu.edu

Phil McMinn
University of Sheffield
Sheffield, United Kingdom
p.mcminn@sheffield.ac.uk

Abstract

Flaky tests produce inconsistent outcomes without code changes, creating major challenges for software developers. An industrial case study reported that developers spend 1.28% of their time repairing flaky tests at a monthly cost of $2,250. This paper reveals that flaky tests often exist in clusters, with co-occurring failures that share the same root causes, which we call *systemic flakiness*. This result suggests that developers can reduce test repair costs by addressing shared root causes, enabling them to fix multiple flaky tests at once rather than tackling them individually. This study represents an inflection point by challenging the deep-seated assumption that flaky test failures are isolated occurrences. We used an established dataset of 10,000 test suite runs from 24 Java projects on GitHub, spanning domains from data orchestration to job scheduling. Using a data set that contains 810 flaky tests, we performed a mixed-method empirical analysis of co-occurring flaky test failures, revealing that systemic flakiness is significant and widespread.

We ran agglomerative clustering of flaky tests based on their failure co-occurrence, showing that 75% of flaky tests across all projects belong to a cluster, with a mean cluster size of 13.5 flaky tests. Instead of requiring 10,000 test suite runs to identify systemic flakiness, this paper demonstrates a lightweight alternative by training machine learning models based on static test case distance measures. Through manual inspection of stack traces, conducted independently by the paper's four authors and resolved through negotiated agreement, we identified intermittent networking issues and instabilities in external dependencies as the predominant causes of systemic flakiness in the chosen open-source projects.

CCS Concepts

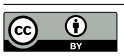
• **Software and its engineering → Software testing and debugging**.

Keywords

Software Testing, Flaky Tests, Systemic Flakiness.

ACM Reference Format:
Owain Parry, Gregory Kapfhammer, Michael Hilton, and Phil McMinn. 2025. Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures. In *Evaluation and Assessment in Software Engineering (EASE '25), June 17–20, 2025, Istanbul, Turkiye.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3756681.3756945

Owain Parry and Phil McMinn are supported by the EPSRC grant "Test FLARE" (EP/X024539/1).



EASE '25, Istanbul, Turkiye

ACM ISBN 979-8-4007-1385-9/25/06
https://doi.org/10.1145/3756681.3756945

1 Introduction

Software developers rely on test cases to identify bugs [31]. However, when test results are unreliable, they lose their value as informative signals and developers may deem them untrustworthy [24]. Practitioners in software testing refer to such unreliable signals as *flaky tests* [42]. While definitions vary [44], a flaky test is generally understood as a test case that can pass or fail unpredictably without changes to its code or the code under test. Flaky tests may arise from concurrency bugs, timing issues, or dependencies on external systems like networks and filesystems [16, 26, 29, 34, 40, 54].

Recent developer surveys and interviews underscore the significant challenges posed by flaky tests [6, 24, 27]. In one survey, 56% of respondents reported encountering flaky tests on a monthly, weekly, or even daily basis [44]. Respondents also strongly agreed that flaky tests disrupt continuous integration [30], reduce productivity, and hinder testing efficiency. The impact of flaky tests is felt across the industry, from large companies like Google and Microsoft [33, 41], to open-source development communities [15, 16]. An industrial case study on the cost of flaky tests in continuous integration found that developers spend up to 1.28% of their time repairing flaky tests, amounting to a monthly cost of $2,250 [36].

This paper's study finds that flaky tests often exist in clusters, with failures that co-occur during the same test suite runs and share the same root causes. We call this phenomenon *systemic flakiness*. It implies that developers can reduce the cost of repairing flaky tests by targeting shared root causes, allowing them to simultaneously fix numerous flaky tests instead of addressing them in isolation.

Surprisingly, systemic flakiness has been neglected in prior research, making this study a key inflection point. Previous studies have relied on simulated failures to assess the impact of flaky tests on milestone techniques in software engineering research, such as fault localization, mutation testing, and automated program repair [14, 55]. However, these simulations misrepresent real-world

flakiness by not accounting for co-occurring flaky test failures. Similarly, prior studies categorized the causes of individual flaky tests without considering systemic flakiness [16, 26, 29, 34, 40, 54]. These studies may have inadvertently reported a skewed distribution of flakiness causes and should be reexamined in light of this study.

This paper's study revisits an existing dataset of 10,000 test suite runs from 24 diverse open-source Java projects that has seen extensive use in prior studies [7, 8, 19, 57]. This dataset contains 810 examples of flaky tests and required over five years of computation time to produce [8]. We leveraged it to perform a mixed-method empirical analysis. This was necessary because no single analysis method can fully capture the complexity of systemic flakiness.

This study combines the following methods:

Method 1: Clustering. To analyze the prevalence of systemic flakiness, we performed agglomerative clustering based on the Jaccard distance between the sets of test suite runs in which flaky tests fail.

Method 2: Prediction. To investigate the feasibility of a lightweight alternative to performing 10,000 test suite runs to identify systemic flakiness, we trained machine learning models to predict the Jaccard distance between pairs of flaky tests using test case distance measures based on static analysis [17, 18].

Method 3: Manual Inspection. To identify systemic flakiness causes, we conducted a qualitative analysis of stack traces and error messages combining manual inspection and negotiated agreement.

This study answers the following research questions:

RQ1: How prevalent is systemic flakiness? Across all projects, 75% of flaky tests fail as part of a cluster, with a mean cluster size of 13.5 flaky tests spanning multiple test classes.

RQ2: How well can machine learning models predict systemic flakiness using static test case distance measures? The prediction accuracy varies across projects, with the best model achieving a mean coefficient of determination of 0.74 when predicting the Jaccard distance between pairs of flaky tests.

RQ3: What causes systemic flakiness? The main causes are networking issues and instabilities in external dependencies.

This paper makes the following main contributions:

Contribution 1: Systemic Perspective. This study introduces a systemic perspective on flaky tests, highlighting the significance of failure co-occurrence, which has been overlooked in prior research.

Contribution 2: Empirical Analysis. This study quantitatively analyzes systemic flakiness using clustering techniques and qualitatively investigates the causes. All the data and results from the empirical study are publicly available in our replication package [1].

Contribution 3: Practical Implications. This study demonstrates the feasibility of leveraging static test case distance measures and machine learning models to predict systemic flakiness, thus offering a viable alternative to exhaustively re-running a test suite.

2 Methodology

This section describes our methodology for answering this paper's three research questions regarding systemic flakiness. We developed Python scripts to automate the key aspects of our analysis, each of which is available in the paper's replication package [1].

2.1 Dataset

We used the dataset created by Alshammari et al. for their evaluation of FlakeFlagger, a machine learning-based technique for automatically detecting flaky tests [8]. They selected 24 Java projects from GitHub that had been used in prior flaky test studies [10, 35]. The projects span a variety of domains from data orchestration to job scheduling. They ran each projects' test suite 10,000 times, requiring over 5 years of computation time. To do so, they created a queue of jobs, where every job represented a single test suite run (running `mvn install`). After each job, they archived all log files, removed all temporary files, and rebooted the machine before proceeding to the next job in the queue. They argued that this approach provided reasonable isolation between test suite runs and suitably simulated how a real build server might compile and test a project.

This dataset contains a JUnit test report for every run of each project's test suite that includes the outcome (pass/fail) of every test case and the full stack trace and error message for test cases that failed. It contains 810 examples of flaky tests (i.e., test cases with an inconsistent outcome) among 22 projects[1]. It also contains the source code of each test case in textual and tokenized form. We selected this dataset due to the vast number of test suite runs, which is very useful for reliably analyzing co-occurring flaky test failures. The available stack traces and source code in this dataset are also useful for effectively answering **RQ2** and **RQ3**.

2.2 Methodology for RQ1: Prevalence

Our scripts performed *agglomerative clustering* of the flaky tests in the projects of the FlakeFlagger dataset using the SciPy library [3]. Agglomerative clustering is a type of clustering technique that starts by treating each data point as its own cluster and then iteratively merges the two closest clusters until all data points are merged into a single cluster [4]. The result of agglomerative clustering is a hierarchy of clusters from which a *concrete clustering* (i.e., an assignment of each data point to a specific cluster) can be extracted by specifying a distance threshold. We selected a clustering approach because it allows us to identify specific instances of systemic flakiness in a project by grouping flaky tests based on their failure co-occurrence. We selected agglomerative clustering specifically because it is does not require us to pre-specify the number of clusters in a project, which could not be known a priori.

Agglomerative clustering requires a distance metric to compare two data points. In the context of this study, a data point is a flaky test and the distance metric needs to capture the extent to which the failures of two flaky tests co-occur. In the FlakeFlagger dataset, each of the 10,000 runs of a project's test suite is associated with a unique numerical ID. As the distance metric to compare two flaky tests, we selected the *Jaccard distance* between the two sets of run IDs in which they fail. The Jaccard distance J between two sets A and B is a measure of how dissimilar they are. It is defined as:

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

For example, suppose one flaky test fails during three test suite runs with IDs $\{52, 901, 5810\}$, and another fails during five test suite runs with IDs $\{52, 901, 1119, 5810, 9402\}$. The length of the union of these two sets is 5 and they have 3 failing run IDs in common, so their Jaccard distance is $1 - 3/5 = 0.4$. In this context, a Jaccard distance of 1 between two flaky tests indicates that there were no

[1] In their original paper, Alshammari et al. reported 811 flaky tests among 23 projects [8]. However, one of the flaky tests did not appear to have any associated stack traces or error messages, so we excluded it from this study.

test suite runs where they both failed, and 0 indicates that they always failed during the same test suite runs. In the latter case, this would imply that the two flaky tests likely share the same root cause and are therefore a manifestation of systemic flakiness. We selected the Jaccard distance because it has a straightforward interpretation and is robust when comparing two flaky tests with vastly different numbers of failing runs. These characteristics are important for this study because we are primarily interested in failure co-occurrence as opposed to the failure rates of individual flaky tests.

Our scripts automatically identified the distance threshold that extracts the concrete clustering with the greatest mean *silhouette score* for each project [50]. For a single flaky test, the silhouette score s evaluates how well it fits within its assigned cluster compared to other clusters. It considers the average Jaccard distance a to the other flaky tests in its assigned cluster and the average distance b to the flaky tests in the nearest other cluster. It is defined as:

$$s = \frac{b - a}{max(a, b)} \tag{2}$$

The mean silhouette score over all the flaky tests in a project evaluates the overall quality of the concrete clustering. It ranges from 1 to -1 where scores above 0.7 indicate strong (i.e., well-defined and compact) clusters, scores above 0.5 indicate reasonable clusters, and scores above 0.25 indicate weak clusters [32]. We assumed that there was no systemic flakiness in a project if there was no possible concrete clustering with a mean silhouette score of at least 0.6, ensuring that the identified clusters are more than merely reasonable without being overly restrictive. We also did not consider clusters containing only a single flaky test (known as *singleton clusters*) as evidence of systemic flakiness because, by definition, they do not identify any failure co-occurrence.

We implemented this aforementioned automated approach to identifying a separate distance threshold for each project for several reasons. Firstly, automatically selecting the distance threshold based on the mean silhouette score removes subjective bias and ensures that the clustering process is consistent and reproducible across projects. Secondly, setting a minimum mean silhouette score of 0.6 ensures that clusters are reasonably well-defined and interpretable. This avoids the risk of identifying spurious instances of systemic flakiness that may undermine the reliability of our analysis. Finally, identifying a distance threshold for each project individually accommodates potential differences in flaky test behaviors, which may depend on the project's size, purpose, or complexity.

2.3 Methodology for RQ2: Prediction

Detecting flaky tests without performing test suite runs has been extensively studied in prior work [12, 19, 47–49, 56]. Techniques to do so typically make use of machine learning models trained on features based on static analysis of test case code, such as specific token occurrences and complexity metrics. Building on this foundation, we set out to evaluate how effectively machine learning models can predict systemic flakiness based on static test case distance measures between pairs of flaky tests. Such distance measures have been applied in prior work to assess test suite diversity [17, 18]. We selected them as features based on the intuition that the more similar two flaky tests are, the more likely their failures will co-occur.

Our scripts trained and evaluated three tree-based ensemble models in the context of a regression task and a classification task using the scikit-learn library [2]. They did this for each project in the FlakeFlagger dataset with at least 10 flaky tests and one cluster. The three models we selected were extra trees, gradient boosting, and random forest [11, 20, 22]. We selected these models because they capture complex nonlinear relationships, perform well even with moderate amounts of training data, and are robust to overfitting thanks to their ensemble nature [53]. For each model, we used the default hyperparameters in scikit-learn. The number of sub-models in the ensemble, arguably the most important hyperparameter, has a default value of 100 in each case. For each pair of flaky tests in a project, the regression task is to predict the Jaccard distance between the two sets of failing run IDs and the classification task is to predict whether they belong to the same cluster.

For the regression task, our scripts evaluated the performance of each model by calculating the mean *coefficient of determination* (R^2) following 5-fold cross validation over the flaky tests in a given project. This is a standard measure for regression tasks that indicates the amount of variance in the target variable (the Jaccard distance) that is explained by the model. It ranges from 1 to negative infinity, where a value of 1 indicates perfect predictions. A model that ignores the input features and always predicts the mean value of the target variable would score 0. For the classification task, our scripts evaluated the performance of each model by calculating the mean *Matthews correlation coefficient* (MCC) following 5-fold *stratified* cross validation. This is a reliable metric for binary classification tasks that is robust to label imbalance [13]. It ranges from 1 to -1, where a value of 1 indicates perfect predictions. A model that always predicts the most common label would score 0.

As features, we selected character-based and set-based distance measures applied to the names and source code of pairs of flaky tests. The character-based measures we selected were the Levenshtein distance, the Damerau-Levenshtein distance, the Jaro distance, and the Jaro-Winkler distance. We also used the normalized variants of the Levenshtein and Damerau-Levenshtein distances. The set-based measures we selected were the Jaccard distance, the Dice distance, and the overlap distance (one minus the overlap coefficient). To calculate the set-based distance measures, our scripts split the names of each flaky test into a set of tokens. (The FlakeFlagger dataset already contains the source code of every test case in tokenized form.) A fully qualified test case name in Java contains the package name, class name, and method name. For example: `package.name.ClassName#methodName`. Our scripts split these into a set of tokens by breaking up the components and splitting the class names and method names based on their capitalization. Applied to the previous example, this would result in the set of unique tokens: `package`, `name`, `class`, and `method`.

We included an additional feature that is a normalized distance measure between two test case names based on their hierarchical structure that we call the *hierarchy distance*. It is calculated by first extracting the *path* of the two test case names by discarding the method name and splitting the remainder into its components. For example, the path of the test case name `foo.bar.Baz#qux` would be the sequence `foo`, `bar`, and `Baz`. Where n is the length of the longer of the two paths and i is the first index where the two paths differ following a zero-based indexing scheme, n if they are identical, or

the length of the shorter of the two paths if one is a prefix of the other, the hierarchy distance is $1 - i/n$. A value of 1 indicates that the two paths are identical and a value of 0 indicates they they differ in their first component. It captures the distance between two test cases within the hierarchical structure of the project test code.

To get a sense of which static test case distance measures are the most important features for systemic flakiness prediction, our scripts calculated *SHAP values* with respect to the regression task and the model with the greatest coefficient of determination [39]. For a given distance measure and pair of flaky tests, the corresponding SHAP value captures the contribution of that distance measure towards the model's prediction of the Jaccard distance between the two sets of failing run IDs. The sum of the SHAP values over each distance measure for a given pair of flaky tests is equal to the model's prediction of their Jaccard distance. For each project, our scripts ranked the distance measures in terms of their mean absolute SHAP value over every pair of flaky tests, thereby giving a general overview of which were the most impactful.

2.4 Methodology for RQ3: Causes

Four authors of this paper engaged in manual inspection of each cluster that our scripts identified for **RQ1**. We randomly allocated two inspecting authors to every project in the dataset with at least one cluster. Each inspector then independently answered a series of questions about every cluster in their allocated projects in a random order. One of the questions was "What are the root causes of the flaky test failures in this cluster?", which directly addresses **RQ3**. Another question was "What actions could a developer take to repair or mitigate the flaky tests in this cluster?". Inspectors were also given the opportunity to offer any miscellaneous insights.

To enable the inspectors to answer these questions for a given cluster, they were able to review the source code of its member flaky tests. Our scripts also randomly sampled stack traces and error messages from their co-occurring failures. We ensured that the inspectors saw a reasonable diversity of stack traces by having our scripts take into account their pairwise Levenshtein distance when sampling. The four inspectors then met and arrived at a collective answer to each question for every cluster they were allocated following a process of negotiated agreement. Negotiated agreement is a process whereby multiple researchers independently analyze a dataset and then systematically compare their findings and discuss discrepancies until all researchers converge on a shared interpretation. It features in the methodologies of previous software engineering studies [30, 45]. One inspector later reviewed the answers to the two questions regarding root causes and possible repairs for each cluster and identified the overarching themes.

2.5 Threats to Validity

While we designed our methodology to rigorously investigate systemic flakiness, we must acknowledge several threats to validity.

Internal Validity. Our clustering analysis relies on the Jaccard distance between sets of failing test suite runs. While this metric effectively captures co-occurrence, other distance metrics might yield different clusters, potentially affecting our results. Moreover, the automated selection of the clustering threshold based on the silhouette score introduces an inherent dependence on this specific metric. While we set a minimum silhouette score of 0.6 to ensure well-defined clusters, borderline cases may have been excluded or misclassified. Errors in the implementation of our scripts could also impact the results. To mitigate this risk, we delegated the most important components to well-established and thoroughly tested third-party libraries such as SciPy and scikit-learn [2, 3].

External Validity. The generalizability of our findings is inherently limited by the chosen dataset. While it represents a broad range of open-source Java projects from diverse domains, the dataset may not capture systemic flakiness behaviors in other programming languages. Moreover, it is possible that the 10,000 test suite runs were not sufficient to manifest all the flaky tests in the projects' test suites and to reliably observe the failure patterns of flaky tests with very low failure rates. As pointed out by the original dataset authors [8] (and implied by other practitioners [28]), this is a general threat to the validity of any empirical study on flaky tests that cannot be totally rectified, but instead only mitigated by performing as many test suite runs as computationally feasible.

Construct Validity. This study assumes that co-occurrence of flaky test failures is a symptom of systemic flakiness. Even though this assumption is well-founded, there may be other factors contributing to failure co-occurrence, including random chance. Additionally, our use of static test case distance measures as predictors of systemic flakiness is based on intuition and prior work on test suite diversity [17, 18]. However, these measures may not fully capture the complexity of relationships between flaky tests.

Conclusion Validity. Our scripts calculated SHAP values to evaluate the importance of features in predicting systemic flakiness. It is important to note that SHAP values only capture a feature's contribution to a model's prediction rather than a feature's value to the prediction task in general. In other words, a feature with a high mean absolute SHAP value with respect to a poorly performing model may not actually indicate anything about that feature's predictive power. The qualitative analysis for **RQ3** is subject to potential bias due to its reliance on subjective human interpretation. To mitigate this, we employed a process of negotiated agreement among the inspectors and randomly allocated clusters to ensure diverse perspectives. However, the findings may still reflect variations in the individual expertise or interpretation of the inspectors.

3 Results

3.1 Answering RQ1: Prevalence

Table 1 shows the results of the agglomerative clustering of the flaky tests in the projects of the FlakeFlagger dataset based on the extent of their failure co-occurrence. The table gives the number of flaky tests and the number of clusters per project after extracting the concrete clustering with the greatest mean silhouette score. For each project that contains at least one cluster, it also gives the number of flaky tests that belong to a cluster, the mean number of flaky tests per cluster, the mean number of distinct test classes per cluster, and finally the mean silhouette score and distance threshold. The total run time required to compute the clusters for every project was 1.8 seconds on a machine with an Intel Core i7-13700 CPU.

Of the 22 projects in the FlakeFlagger dataset that contain at least one flaky test, 10 (45%) contain at least one cluster. The remainder have either only a single flaky test or no possible concrete clustering

Table 1: Results of the agglomerative clustering of flaky tests based on their failure co-occurrence. The table gives the number of flaky tests (Flaky Tests) and the number of clusters (Clusters). For projects with at least one cluster, it also gives the number of flaky tests that belong to a cluster (Flaky in Cluster), the mean number of flaky tests per cluster (Mean Size), the mean number of test classes per cluster (Mean Classes), the mean silhouette score (Silhouette), and the distance threshold (Threshold).

Project Name	Flaky Tests	Clusters	Flaky in Cluster	Mean Size	Mean Classes	Silhouette	Threshold
activiti-activiti	32	0	-	-	-	-	-
Alluxio-alluxio	116	1	113	113.00	16.00	0.88	0.52
apache-ambari	52	2	50	25.00	1.50	0.96	0.00
apache-hbase	145	9	135	15.00	4.78	0.91	0.09
apache-httpcore	22	0	-	-	-	-	-
apache-incubator-dubbo	19	0	-	-	-	-	-
doanduyhai-Achilles	4	0	-	-	-	-	-
elasticjob-elastic-job-lite	3	1	2	2.00	1.00	0.67	0.00
hector-client-hector	33	1	31	31.00	7.00	0.94	0.00
jknack-handlebars.java	1	0	-	-	-	-	-
joel-costigliola-assertj-core	1	0	-	-	-	-	-
kevinsawicki-http-request	18	3	18	6.00	1.00	0.99	0.01
ninjaframework-ninja	1	0	-	-	-	-	-
orbit-orbit	7	0	-	-	-	-	-
qos-ch-logback	22	0	-	-	-	-	-
spring-projects-spring-boot	163	8	154	19.25	4.12	0.94	0.01
square-okhttp	100	10	74	7.40	1.10	0.74	0.00
tootallnate-java-websocket	23	0	-	-	-	-	-
undertow-io-undertow	7	0	-	-	-	-	-
wildfly-wildfly	23	6	18	3.00	1.00	0.78	0.00
wro4j-wro4j	16	4	11	2.75	2.00	0.64	0.11
zxing-zxing	2	0	-	-	-	-	-
Overall	810	45	606	13.47	2.91	-	-

with a mean silhouette score of at least 0.6. There are 810 flaky tests and 45 clusters between the 22 projects. Of the 810 flaky tests, 606 (75%) belong to a cluster. The mean number of flaky tests per cluster varies considerably between projects. The mean size over the 45 clusters is 13.5 flaky tests. On average, clusters contain flaky tests from 2.9 distinct test classes. These results indicate that systemic flakiness is widespread and extends beyond test class boundaries.

Conclusion for RQ1. Systemic flakiness is a widespread and significant phenomenon. There are 45 clusters between the 22 projects in the FlakeFlagger dataset that contain flaky tests. Of the 810 flaky tests in the dataset, 606 (75%) belong to a cluster. The mean cluster size is 13.5 flaky tests.

3.2 Answering RQ2: Prediction

Table 2 presents the effectiveness of three machine learning models at using static test case distance measures to predict systemic flakiness in projects with at least 10 flaky tests and one cluster. For each type of model (extra trees, gradient boosting, and random forest), the table gives the performance for the regression task using the coefficient of determination (R^2) and the classification task using the Matthews correlation coefficient (MCC). For each pair of flaky tests in a project, the regression task is to predict the Jaccard distance between the two sets of failing run IDs and the classification task is to predict whether they belong to the same cluster. The total run time required to train and evaluate every model for each project for both tasks was 210 seconds with an Intel Core i7-13700 CPU.

For the regression task, the extra trees model has the greatest mean performance in terms of R^2 at 0.74. This reveals that, on average, 74% of the variance in the Jaccard distance over every pair of flaky tests is explained by the model using the static test case distance measures described in Section 2.3 as features. There is inconsistency in the per-project performance of all three models. Focusing on the extra trees model, the performance is acceptable for most projects, achieving an R^2 of or above 0.8 for 6 out of 9 projects. It is particularly effective for apache-ambari, spring-projects-spring-boot, and wildfly-wildfly, where it achieves an R^2 above 0.9. The performance is quite poor for the projects square-okhttp and wro4j-wro4j where the model fails to explain even half of the variance. This pattern is reflected by the other two models to varying extents, indicating that it is due to properties of these projects and the features rather than the extra trees model specifically.

For the classification task, the extra trees model has the greatest mean performance in terms of MCC at 0.74. The per-project performance for this task appears roughly correlated with that for the regression task. This is unsurprising, given they both use the same features and evaluate the ability to predict systemic flakiness.

Table 3 gives the ranks of each static test case distance measure in terms of their mean absolute SHAP value with respect to the

Table 2: The performance of three machine learning models for predicting systemic flakiness using static test case distance measures in projects with at least 10 flaky tests and one cluster. The table gives the performance for the regression task using the coefficient of determination (R^2) and the classification task using the Matthews correlation coefficient (MCC).

	Extra Trees		Gradient Boosting		Random Forest	
Project Name	**Regression (R^2)**	**Classification (MCC)**	**Regression (R^2)**	**Classification (MCC)**	**Regression (R^2)**	**Classification (MCC)**
Alluxio-alluxio	0.51	0.41	0.27	0.24	0.48	0.39
apache-ambari	0.98	0.99	0.97	0.98	0.96	0.97
apache-hbase	0.80	0.84	0.63	0.74	0.77	0.83
hector-client-hector	0.87	0.95	0.76	0.85	0.78	0.88
kevinsawicki-http-request	0.88	0.77	0.85	0.65	0.82	0.66
spring-projects-spring-boot	0.95	0.96	0.87	0.92	0.93	0.96
square-okhttp	0.36	0.29	0.37	0.19	0.38	0.27
wildfly-wildfly	0.92	0.87	0.86	0.87	0.88	0.75
wro4j-wro4j	0.37	0.62	0.36	0.23	0.38	0.35
Mean	0.74	0.74	0.66	0.63	0.71	0.67

Table 3: The ranks of the 21 static test case distances measures in terms of their mean absolute SHAP value with respect to the regression task and the extra trees model. Lower ranks indicate that the distance measure has a greater mean absolute SHAP values and was thus a more important feature. See Section 2.3 for the details about the distance measures and the SHAP values.

		Importance Rank									
Distance	**Applied to**	***Alluxio-alluxio***	***apache-ambari***	***apache-hbase***	***hector-client...***	***kevinsawicki-http...***	***spring-projects...***	***square-okhttp***	***wildfly-wildfly***	***wro4j-wro4j***	**Mean**
Hierarchy	Name	1	1	1	9	21	1	1	1	3	4.33
Overlap	Name	9	2	2	10	4	6	10	18	2	7.00
Levenshtein	Name	3	16	14	5	11	9	2	8	12	8.89
Normalized Levenshtein	Name	13	3	3	19	15	7	9	3	8	8.89
Overlap	Code	6	10	15	3	1	2	14	10	20	9.00
Damerau-Levenshtein	Name	2	14	16	7	12	8	3	6	14	9.11
Dice	Name	7	5	9	17	14	11	6	9	5	9.22
Normalized Damerau-Levenshtein	Name	18	4	4	16	10	13	5	7	7	9.33
Jaro	Name	15	6	7	15	17	12	11	2	1	9.56
Normalized Compression	Name	11	9	5	11	16	18	4	5	9	9.78
Jaccard	Name	8	7	12	18	6	10	13	11	6	10.11
Levenshtein	Code	5	12	8	14	2	14	7	12	21	10.56
Damerau-Levenshtein	Code	4	11	10	13	3	16	8	13	19	10.78
Dice	Code	17	13	13	1	7	3	15	21	16	11.78
Jaccard	Code	16	15	17	2	5	4	16	20	15	12.22
Jaro-Winkler	Name	20	8	11	20	18	19	12	4	4	12.89
Normalized Compression	Code	10	17	6	21	13	5	17	16	13	13.11
Normalized Levenshtein	Code	12	18	19	8	8	17	18	17	18	15.00
Normalized Damerau-Levenshtein	Code	14	21	18	12	9	15	19	19	17	16.00
Jaro	Code	19	20	20	4	20	20	21	15	11	16.67
Jaro-Winkler	Code	21	19	21	6	19	21	20	14	10	16.78

Table 4: Cause theme frequencies from the qualitative analysis of the 45 clusters. The totals sum to greater than 45 because some clusters belong to multiple themes.

Project Name	Networking	External Dependency	Filesystem Pollution	Timeout	Unknown	System Clock
Alluxio-alluxio	1	0	0	0	0	0
apache-ambari	2	0	0	0	0	0
apache-hbase	6	0	5	3	0	0
elasticjob-elastic-job-lite	0	0	0	0	0	1
hector-client-hector	0	1	0	0	0	0
kevinsawicki-http-request	3	0	0	0	0	0
spring-projects-spring-boot	1	6	0	0	1	0
square-okhttp	10	0	0	0	0	0
wildfly-wildfly	1	5	0	1	0	0
wro4j-wro4j	1	2	0	0	1	0
Total	25	14	5	4	2	1

regression task and extra trees model per project. A rank of 1 indicates that the distance measure has the greatest mean absolute SHAP value and was thus the feature with the greatest impact on the model's prediction. A rank of 21 indicates the opposite. The table also gives the mean rank for each feature over every project. According to the mean ranks, the hierarchy distance was the most important feature and distance measures applied to the names of test cases were generally more important than those applied to the source code. However, the per-project ranks for each feature vary significantly. In the case of the hierarchy distance, it was the least important feature for the kevinsawicki-http-request project despite being the most important feature on average across all projects.

Conclusion for RQ2. Machine learning models can predict systemic flakiness using static test case distance measures to varying extents across projects. The extra trees model has the greatest mean performance at the regression task, achieving an R^2 of 0.74, and the classification task, achieving an MCC of 0.74. On average, the hierarchy distance is the most important feature and distance measures applied to the names of test cases are generally more important than those applied to the code.

3.3 Answering RQ3: Causes

Table 4 summarizes the results of our qualitative analysis of the causes of the 45 clusters in the FlakeFlagger dataset. It gives the frequencies of each cause theme that we identified. Clusters for which we were unable to identify the cause are represented by the *Unknown* theme. This table's totals sum to greater than the number of clusters because some clusters belong to multiple themes.

The most common theme is *Networking*. This represents clusters where an intermittent networking issue causes the failure of a group of flaky tests during a single test suite run. In these instances, the test case tries to establish a connection over the network or executes production code that does. Examples of such networking issues include DNS resolution failures, an unreachable network, and timeouts due to the network taking longer than expected to process requests. Generally speaking, we found clusters in this category to be the largest and most diverse in terms of their member flaky tests. This is probably because an intermittent networking issue is likely to impact any test case that requires a functional network connection regardless of its purpose or which part of the production code that it tests. See Figure 1 for an example of a *Networking* cluster. Please refer to the replication package [1] for more examples.

The second most common theme is *External Dependency*. Clusters of this theme contain groups of flaky tests that all depend on some external service, library, or other artifact that is outside the control of the software under test. If the external service is unavailable or is exhibiting unexpected behavior during a particular test suite run, then all the flaky tests that depend upon it are likely to fail at the same time. In one cluster of the spring-projects-spring-boot project, the external dependency issue appeared to be related to non-determinism in the version of Spring Framework that was being installed when building the project before each test suite run. In another cluster of the wildfly-wildfly project, flaky tests were reliant on an external web server that was intermittently returning 500 errors, which is an HTTP status code indicating a server-side error. We did not consider this to be an instance of the *Networking* theme because the server encountered some sort of intermittent problem and the network itself did not fail.

Less common themes are *Filesystem Pollution*, *Timeout*, and *System Clock*. In *Filesystem Pollution*, clusters are caused by one or more test cases modifying the filesystem (e.g., creating a directory) and then failing in such a way that they omit to perform proper clean up procedures. This triggers the failure of a group of subsequent test cases due to the unexpected state of the filesystem. In *Timeout*, a hard coded time limit for some event to occur triggers the failure of a group of flaky tests. This typically co-occurs with the *Networking* theme where the time limit concerns a response from a server. We observed only a single cluster under the *System Clock* theme, where a small cluster of flaky tests that made direct use of the system clock failed at the same time.

Conclusion for RQ3. Following qualitative analysis of the causes of the 45 clusters, the most common theme is *Networking*. This theme represents clusters where an intermittent networking issue causes the failure of a group of flaky tests during a single test suite run. The second most common theme is *External Dependency*. Clusters assigned to this theme contain groups of flaky tests that all depend on some external service, library, or other artifact that is outside the control of the software under test.

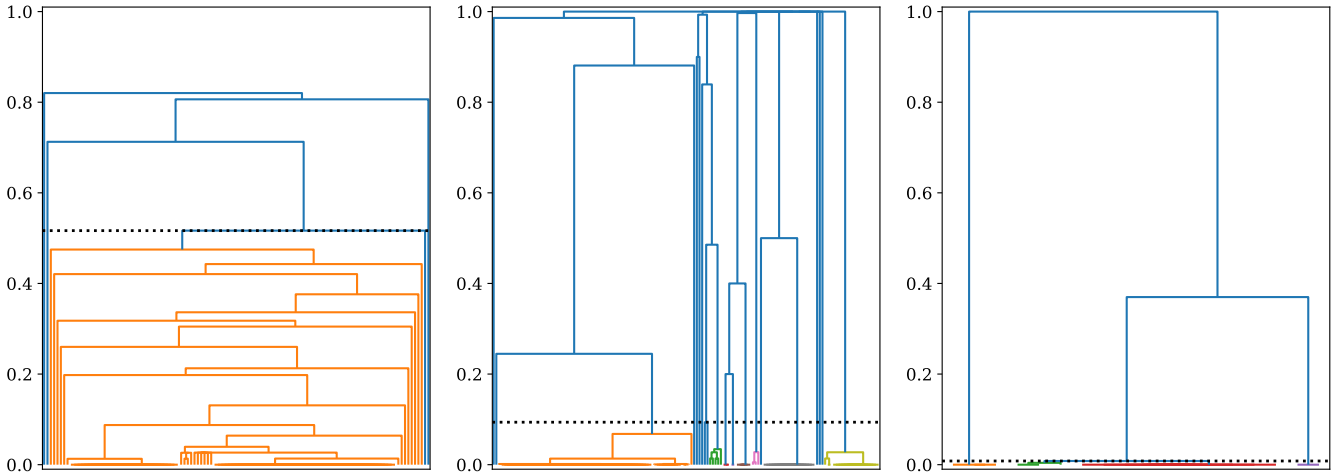
4 Discussion

4.1 Distance Thresholds

Figure 2 illustrates the hierarchy of the clusters, prior to extracting a concrete clustering, in the form of a dendrogram for three projects. Each dendrogram represents a tree structure, where leaves represent flaky tests and branches represent how they are progressively clustered based on their failure co-occurrence. The vertical

```
@Test(expected = BadCredentialsException.class)
public void testBadCredential() throws Exception {
    Authentication authentication = new UsernamePasswordAuthenticationToken("notFound", "wrong");
    authenticationProvider.authenticate(authentication);
}
@Test
public void testAuthenticate() throws Exception {
    assertNull("User already exists in DB", userDAO.findLdapUserByName("allowedUser"));
    Authentication authentication = new UsernamePasswordAuthenticationToken("allowedUser", "password");
    Authentication result = authenticationProvider.authenticate(authentication);
    assertTrue(result.isAuthenticated());
    assertNotNull("User was not created", userDAO.findLdapUserByName("allowedUser"));
    result = authenticationProvider.authenticate(authentication);
    assertTrue(result.isAuthenticated());
}
```

Figure 1: The source code of two flaky test cases from the apache-ambari project that form a *Networking* cluster. They both failed after calling the `authenticate` method during the exact same 8 test suite runs out of the 10,000 total runs. In both circumstances, the root exception was `java.net.ConnectException: Connection refused (Connection refused)`.



(a) Alluxio-alluxio **(b) apache-hbase** **(c) kevinsawicki-http-request**

Figure 2: Dendrograms for three projects illustrating the hierarchy of the clusters prior to extracting a concrete clustering. The vertical axis shows the Jaccard distance at which clusters are merged (see Equation 1). The dotted line represents the distance threshold that produces the concrete clustering with the greatest mean silhouette score (see Equation 2).

axis shows the Jaccard distance at which clusters are merged (see Equation 1), with lower branches representing more similar sets of failing run IDs between flaky tests. The dotted line represents the distance threshold that produces the concrete clustering with the greatest mean silhouette score (see Equation 2), as identified by our scripts. The clusters are color coded, with blue representing individual flaky tests that did not make it into a non-singleton cluster. Other clusters are assigned distinct colors to visually differentiate them, though the specific colors do not carry inherent meaning.

For Alluxio-alluxio, there are many branches distributed fairly evenly as the distance increases up to about 0.8. This indicates that the sets of failing run IDs of the flaky tests in this project are rather diverse. There is a single cluster for this project and the distance threshold is 0.52, which indicates a cluster of flaky tests with relatively heterogeneous, but still related, sets of failing run IDs. For kevinsawicki-http-request, the branches are mainly concentrated at the very bottom of the dendrogram. This indicates flaky tests with very similar sets of failing run IDs. There are three clusters for this project and the distance threshold is 0.01, which indicates clusters of flaky tests with almost identical sets of failing run IDs. The dendrogram for apache-hbase represents a situation somewhere in between. The threshold for five projects is 0, indicating clusters of flaky tests with strictly identical sets of failing run IDs.

4.2 Repairs and Mitigations

One of the questions the inspectors answered about each cluster during the qualitative analysis was "What actions could a developer take to repair or mitigate the flaky tests in this cluster?". Table 5 gives the frequencies of the general themes in the collective answers to this question. Generally speaking, it was difficult for the

Table 5: Repair/mitigation theme frequencies from our qualitative analysis of the 45 clusters. The totals sum to greater than 45 because some clusters belong to multiple themes.

Project Name	Look Before You Leap	Unknown	Mocking	Better Setup/Teardown	Better Error Checking	Avoid Networking
Alluxio-alluxio	1	0	1	0	0	0
apache-ambari	1	0	0	2	0	0
apache-hbase	1	1	4	2	3	0
elasticjob-elastic-job-lite	0	0	1	0	0	0
hector-client-hector	0	1	0	0	0	0
kevinsawicki-http-request	1	0	0	0	0	3
spring-projects-spring-boot	1	7	1	0	0	0
square-okhttp	10	0	0	0	0	0
wildfly-wildfly	5	1	0	0	0	0
wro4j-wro4j	0	4	0	0	0	0
Total	20	14	7	4	3	3

inspectors to confidently answer this question because they were not familiar with the intricacies of the chosen software projects.

The most common theme by far was *Look Before You Leap*. This represents clusters of flaky tests that could be repaired or at least mitigated by checking the status of some external system or resource. Examples include checking for the existence of a directory and confirming that a server is running. The second most common theme was *Mocking*. Clusters of this theme could be addressed through proper mocking of some third-party library or external service [52]. For example, the cluster of the *System Clock* cause theme could have been mitigated by mocking the system clock.

In the *Better Setup/Teardown* theme, inspectors identified possible improvements to the setup and teardown methods that are executed before and after test cases. In a recent developer survey, respondents rated issues in setup and teardown methods to be the most common causes of flaky tests [44]. In *Better Error Checking*, clusters of flaky tests failed with nondescript error messages such as `NullPointerException`, caused by intermittent problems deep in the call stack. Inspectors suggested more comprehensive error checking to facilitate debugging, making this theme more of a mitigation strategy than a genuine repair. Finally, in the *Avoid Networking* theme, inspectors determined that the underlying program logic being evaluated by the flaky tests could have been tested independently of any networking, which was the cause of the flakiness.

4.3 Implications and Future Directions

This paper's empirical study is the first to characterize systemic flakiness and as such represents an inflection point in flaky test research. The findings have major implications for developers and researchers, and open up a myriad of avenues for future work.

Cost and Developer Impact. This study found that the mean cluster size over all projects was 13.5 flaky tests. An industrial case study on the cost of flaky tests in continuous integration found that developers allocated up to 1.28% of their time to repairing flaky tests, translating to a monthly cost of $2,250 [36]. By recognizing systemic flakiness, developers can achieve significant cost and time savings by resolving underlying root causes that simultaneously fix multiple flaky tests, rather than inefficiently debugging and repairing them in isolation. Future studies should focus on developing automated techniques to detect and triage systemic flakiness in continuous integration pipelines, thereby reducing the cost of flakiness.

Impact on Testing Techniques. This study found that 75% of flaky tests across all projects fail as part of a cluster. Prior studies evaluated the negative impacts of simulated flakiness on fault localization, mutation testing, and automated program repair [14, 55]. These studies did not consider systemic flakiness and did not simulate clusters of flaky tests with co-occurring failures. Therefore, the impact of flaky tests on these techniques may be misrepresented. Future studies should revisit these assessments using a more realistic simulation model that accounts for systemic flakiness.

Machine Learning for Systemic Flakiness Prediction. This study found that machine learning models can predict systemic flakiness using static test case distance measures. Previous studies have evaluated machine learning-based techniques to classify individual test cases as flaky or not [8, 12, 19, 25, 43, 46–49, 56]. By not considering systemic flakiness, these techniques do not benefit from valuable contextual features, such as historical patterns of failure co-occurrence. Without this information, techniques cannot identify inter-test relationships, leading to predictions that focus on isolated flaky tests rather than underlying root causes. This may limit a technique's generalizability across projects, which is a well-known limitation [5]. Future studies should explore integrating systemic flakiness prediction into existing flaky test classification techniques and evaluate its impact on cross-project generalization.

Feature Engineering. This study found that the hierarchy distance was the most important feature for systemic flakiness prediction, and that distance measures applied to the names of test cases were generally more important than those applied to the source code. Prior studies have established that the inclusion of dynamic features enhances machine learning-based flaky test detection [8, 43]. Dynamic test case distance measures, such as the Jaccard distance between coverage profiles, may capture additional signals that static test case distance measures miss. Future studies should refine the feature set and explore additional metrics.

Causes and Mitigation Strategies. This study identified intermittent networking issues and instabilities in external dependencies as predominant causes of systemic flakiness through manual inspection. In contrast, previous studies that categorized the causes of individual flaky tests generally rated asynchronous operations and concurrency as the leading causes [16, 26, 29, 34, 40, 54]. By overlooking systemic flakiness, these studies may have unintentionally presented a skewed distribution of flaky test causes. Future studies should conduct larger-scale empirical analyses of the causes of systemic flakiness across different programming languages and testing frameworks to rectify this. This study also found that clusters contain flaky tests from 2.9 distinct test classes on average. This suggests that developers should not only take greater care to isolate

test cases from environmental variability but should also ensure that test classes are properly decoupled. Future studies should focus on automated techniques to assist developers in this regard.

Automating Root Cause Analysis. This study involved manual inspection of each cluster, which was a time-consuming process and in some cases did not identify any causes. Future studies should leverage artificial intelligence methods to automatically identify systemic flakiness causes by analysis of test code and stack traces.

Developer Perception. This study was based on quantitative analysis of test execution data and manual inspection conducted by the authors. Industrial software developers were not directly involved in the methodology. Understanding how developers currently perceive and address systemic flakiness is crucial for designing techniques to address it. Future studies should conduct developer surveys and interviews to assess whether developers are aware of systemic flakiness and, if so, how they deal with it.

Benchmarking and Dataset Creation. Even though this study relied on an established dataset of flaky test failures, no existing datasets explicitly capture systemic flakiness. Future studies should focus on creating benchmark datasets that annotate failure co-occurrence, enabling further research into systemic flakiness detection and mitigation. It would be beneficial for the purposes of generalizability if projects written in multiple programming languages were represented. These datasets could be used to evaluate new machine learning models and continuous integration strategies.

5 Related Work

Golagha et al. proposed a technique to cluster failing hardware-in-the-loop tests based on non-code-based features in the absence of coverage data, which they argued is difficult to acquire in that domain [23]. Both their study and this study grouped failing test cases using agglomerative clustering, but with different aims. The aim in their study was to reduce manual debugging effort while still identifying as many bugs as possible by selecting only a single representative failing test case from each cluster for developers to review. The aim in this study was to identify instances of systemic flakiness by clustering flaky tests based on failure co-occurrence.

An et al. proposed a machine learning-based technique to automatically identify if a pair of failing test cases share the same root cause in the continuous integration pipeline of SAP HANA [9]. Both their study and this study evaluated the capability of machine learning models to predict whether test failures share a common underlying cause based on pairwise similarity/distance measures, but with different aims. The aim in their study was to reduce redundant bug reports and manual debugging effort. The aim in this study was to evaluate the feasibility of a lightweight alternative to performing 10,000 test suite runs to identify systemic flakiness.

Prior studies on flaky test detection have explored machine learning techniques to identify flaky tests without requiring thousands of reruns [8, 12, 19, 25, 43, 46–49, 56]. Pinto et al. investigated whether flaky tests have a distinct "vocabulary" of identifiers and keywords, training machine learning classifiers on vocabulary-based features extracted from test case bodies [47]. Their features included occurrences of whole identifiers and their components, along with complexity metrics such as lines of code. Alshammari et al. developed and evaluated FlakeFlagger, selecting 16 test case features as potential indicators of flakiness. These included eight boolean features capturing test smells [21] and several numeric features, such as lines of code, number of assertions, and production code coverage. This study differs from these prior studies in that we apply machine learning to predict systemic flakiness, using static test case distance measures to estimate the Jaccard distance between the sets of failing run IDs of pairs of flaky tests.

Several previous studies have categorized flaky tests by their root causes via manual inspection [16, 26, 29, 34, 40, 54]. Luo et al. categorized 201 commits that repaired flaky tests from 51 projects of varying size and language [40]. They identified asynchronous calls, concurrency bugs, and test order dependencies as the most common causes. Eck et al. asked 21 software developers to categorize 200 flaky tests that they had previously repaired [16]. They also identified concurrency bugs and asynchronous calls as the most common causes, corroborating the findings of Luo et al., alongside overly restrictive assertion ranges. While these previous studies focus on the causes of individual flaky tests, this paper's study examines clusters of flaky tests, revealing systemic causes such as networking issues and instabilities in external dependencies.

A significant body of work has addressed *order-dependent* flaky tests, whose outcome depends on the execution order of test cases [35, 37, 38, 51, 58]. This is typically caused by side effects left behind by previously executed test cases in the global program state (e.g., static fields in Java) or in the filesystem. This study highlighted filesystem pollution as one of the possible causes of systemic flakiness. However, reordering the test cases, which prior studies typically perform to identify order-dependent flaky tests [35, 59], was never part of the methodology of this study or of the study that produced the FlakeFlagger dataset [8]. Therefore, systemic flakiness is a concept that is clearly distinct from order-dependent flakiness.

6 Conclusion and Future Work

This paper established systemic flakiness as a widespread and significant phenomenon. Through agglomerative clustering, we found that 75% of flaky tests in the dataset belong to a cluster, indicating that flaky tests frequently fail together. We demonstrated that machine learning models can predict systemic flakiness using static test case distance measures. Extra trees was the best performing model on average, achieving an R^2 of 0.74 when predicting the Jaccard distance between the sets of failing run IDs of pairs of flaky tests. The hierarchy distance measure was the most important feature on average in terms of mean absolute SHAP value. Manual inspection of flaky test clusters revealed that systemic flakiness is primarily driven by intermittent networking issues and instabilities in external dependencies. These results emphasize that flaky tests often share causes that transcend individual test case logic.

The prevalence of systemic flakiness has important implications for developers because it shows that they can simultaneously repair multiple flaky tests by addressing the underlying shared root causes. It also has important implications for research because it challenges the assumption that flaky test failures are isolated occurrences.

As part of future work, we will investigate the directions identified in Section 4.3. We specifically plan to extend our study by evaluating a larger set of projects from multiple programming languages. In doing so, we will be able to assess the generalizability of our findings beyond Java projects. This will also result in a comprehensive dataset specifically for studying systemic flakiness.

References

[1] 2025. Replication Package, https://doi.org/10.5281/zenodo.15267575.

[2] 2025. scikit-learn: machine learning in Python — scikit-learn 1.5.2 documentation, https://scikit-learn.org/1.5/index.html.

[3] 2025. SciPy documentation — SciPy v1.14.1 Manual, https://docs.scipy.org/doc/scipy-1.14.1/index.html.

[4] M. R. Ackermann, J. Blömer, D. Kuntze, and C. Sohler. 2014. Analysis of Agglomerative Clustering. *Algorithmica* 1, 2 (2014), 184–215.

[5] A. Afeltra, A. Cannavale, F. Pecorelli, V. Pontillo, and F. Palomba. 2024. A Large-Scale Empirical Investigation Into Cross-Project Flaky Test Prediction. *IEEE Access* 12 (2024), 131255–131265.

[6] A. Ahmad, O. Leifler, and K. Sandahl. 2021. Empirical Analysis of Practitioners' Perceptions of Test Flakiness Factors. *Software Testing Verification and Reliability* 31, 8 (2021), 1–24.

[7] A. Alshammari, P. Ammann, M. Hilton, and J. Bell. 2024. A Study of Flaky Failure De-Duplication to Identify Unreliably Killed Mutants. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 257–262.

[8] A. Alshammari, C. Morris, M. Hilton, and J. Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[9] G. An, J. Yoon, J. Sohn, j. Hong, D. Hwang, and S. Yoo. 2022. Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 65–74.

[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 433–444.

[11] L. Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[12] B. Camara, M. Silva, A. Endo, and Vergilio S. 2021. What is the Vocabulary of Flaky Tests? An Extended Replication. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 444–454.

[13] G. Chicco, D. Jurman. 2020. The Advantages of the Matthews Correlation Coefficient (MCC) Over F1 Score and Accuracy in Binary Classification Evaluation. *BMC Genomics* 21, 6 (2020), 1471–2164.

[14] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman. 2022. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 982–994.

[15] T. Durieux, C. L. Goues, M. Hilton, and R. Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 254–264.

[16] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 830–840.

[17] I. Elgendy, R. Hierons, and P. McMinn. 2024. Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites. In *Proceedings of the International Conference on Automation of Software Test (AST)*. 171–181.

[18] I. Elgendy, R. Hierons, and P. McMinn. 2025. A Systematic Mapping Study of the Metrics, Uses and Subjects of Diversity-Based Testing Techniques. *Software Testing, Verification and Reliability* 35, 2 (2025), e1914.

[19] S. Fatima, T. Ghaleb, and L. Briand. 2022. Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests. *Transactions on Software Engineering* (2022), 1–17.

[20] J. H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232.

[21] V. Garousi and B. Küçük. 2018. Smells in Software Test Code: A Survey of Knowledge in Industry and Academia. *Journal of Systems and Software* 138 (2018), 52–81.

[22] P Geurts, D. Ernst, and L. Wehenkel. 2006. Extremely Randomized Trees. *Machine Learning* 63, 1 (2006), 3–42.

[23] M. Golagha, C. Lehnhoff, A. Pretschner, and H. Ilmberger. 2019. Failure Clustering Without Coverage. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 134–145.

[24] M. Gruber and G. Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need to Address It. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

[25] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser. 2023. Practical Flaky Test Prediction using Common Code Evolution and Test History Data. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

[26] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

[27] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon. 2022. A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

[28] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23.

[29] N. Hashemi, A. Tahir, and S. Rasheed. 2022. An Empirical Study of Flaky Tests in JavaScript. In *International Conference on Software Maintenance and Evolution (ICSME)*. 24–34.

[30] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. 197–207.

[31] G. M. Kapfhammer. 2004. Software Testing. In *The Computer Science Handbook*.

[32] L. Kaufman and P. J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons.

[33] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

[34] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1471–1482.

[35] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 312–322.

[36] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jurgens. 2024. Cost of Flaky Tests in Continuous Integration: An Industrial Case Study. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 329–340.

[37] C. Li, C. Zhu, and A. Wang, W. Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1881–1892.

[38] S. Li and A. Shi. 2022. Evolution-Aware Detection of Order-Dependent Flaky Tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 114–125.

[39] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S. Lee. 2020. From Local Explanations to Global Understanding with Explainable AI for Trees. *Nature Machine Intelligence* 2, 1 (2020), 2522–5839.

[40] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. 643–653.

[41] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-Scale Continuous Testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 233–242.

[42] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–74.

[43] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 93–104.

[44] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 253–262.

[45] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?. In *Proceedings of the International Conference on Automation of Software Test (AST)*. 160–164.

[46] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2023. Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models. *Empirical Software Engineering* 28, 72 (2023).

[47] G. Pinto, B. Miranda, S. Dissanayake, M. D. Amorim, C. Treude, A. Bertolino, and M. D'amorim. 2020. What is the Vocabulary of Flaky Tests?. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 492–502.

[48] V. Pontillo, F. Palomba, and F. Ferrucci. 2022. Static Test Flakiness Prediction: How Far Can We Go? *Empirical Software Engineering* (2022), 325–327.

[49] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, and X. Mao. 2022. PEELER: Learning to Effectively Predict Flakiness without Running Tests. In *International Conference on Software Maintenance and Evolution (ICSME)*. 257–268.

[50] K. R. Shahapure and C. Nicholas. 2020. Cluster Quality Analysis Using Silhouette Score. In *Proceedings of the International Conference on Data Science and Advanced Analytics (DSAA)*. 747–748.

[51] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 545–555.

[52] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. 2017. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 402–412.

[53] S. Uddin and H. Lu. 2024. Confirming the Statistically Significant Superiority of Tree-Based Machine Learning Algorithms Over Their Counterparts for Tabular Data. *Plos One* 19, 4 (2024), e0301541.

[54] A. Vahabzadeh, A. A. Fard, and A. Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 101–110.

[55] B. Vancsics, T. Gergely, and A. Beszédes. 2020. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness. In *Proceedings of the International Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. 28–35.

[56] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino. 2021. Know Your Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021), 76119–76134. Issue 4.

[57] J. Wang, Y. Lei, M. Li, G. Ren, H. Xie, S. Jin, J. Li, and J. Hu. 2024. Flakyrank: Predicting Flaky Tests Using Augmented Learning to Rank. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 872–883.

[58] R. Wang, Y. Chen, and W. Lam. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C)*. 120–124.

[59] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 385–396.