Can We Classify Flaky Tests Using Only Test Code? An LLM-Based Empirical Study

Alexander Berndt *
Heidelberg University
alexander.berndt@uni-heidelberg.de

Vekil Bekmyradov
SAP
vekil.bekmyradov@sap.com

Rainer Gemulla
University of Mannheim
rgemulla@uni-mannheim.de

Marcus Kessel
University of Mannheim
marcus.kessel@uni-mannheim.de

Thomas Bach
SAP
thomas.bach03@sap.com

Sebastian Baltes
Heidelberg University
sebastian.baltes@uni-heidelberg.de

***Abstract*—Flaky tests yield inconsistent results when they are repeatedly executed on the same code revision. They interfere with automated quality assurance of code changes and hinder efficient software testing. Previous work evaluated approaches to train machine learning models to classify flaky tests based on identifiers in the test code. However, the resulting classifiers have been shown to lack generalizability, hindering their applicability in practical environments. Recently, pre-trained Large Language Models (LLMs) have shown the capability to generalize across various tasks. Thus, they represent a promising approach to address the generalizability problem of previous approaches.**

In this study, we evaluated three LLMs (two general-purpose models, one code-specific model) using three prompting techniques on two benchmark datasets from prior studies on flaky test classification. Furthermore, we manually investigated 50 samples from the given datasets to determine whether classifying flaky tests based only on test code is feasible for humans.

Our findings indicate that LLMs struggle to classify flaky tests given only the test code. The results of our best prompt-model combination were only marginally better than random guessing. In our manual analysis, we found that the test code does not necessarily contain sufficient information for a flakiness classification. Our findings motivate future work to evaluate LLMs for flakiness classification with additional context, for example, using retrieval-augmented generation or agentic AI.

***Index Terms*—software testing, flaky tests, large language models, classification, negative results**

I. Introduction

Flaky tests yield inconsistent results when they are repeatedly executed on the same code revision [1]. They interfere with continuous integration pipelines, hamper efficient automated software testing, and diminish developers' trust in the reliability of test results [2]. Previous work has proposed various approaches to detect and fix flaky tests [1]–[9]. Typically, flaky tests can be detected by repeatedly executing a test and analyzing its results. However, repeatedly executing tests is costly with respect to computational resources.

A common approach to detect flaky tests without repeated test execution is to train a machine learning model on the task of flakiness classification [3], [7], [9]–[18]. For example, Pinto et al. proposed training machine learning classifiers on bag-of-words representations of the test code with promising results [13]. Additional studies evaluated various models, training approaches, and evaluation setups for the task of flakiness classification based on test code [7], [15]–[18]. Although the resulting classifiers generally achieved strong results on established benchmarks, evaluations conducted under conditions resembling real-world usage indicated limited generalizability caused by an overfitting of the classifiers to the test code vocabulary in the training set [7], [15], [16].

To gain a deeper understanding of flakiness classification based on the test code, in this study, we investigated the following research questions:

RQ1: What is the performance of LLMs without additional fine-tuning in flaky test classification based on the test code...

(1.1) ...with a zero-shot prompt?

(1.2) ...with a zero-shot chain-of-thought prompt?

(1.3) ...with a few-shot chain-of-thought prompt?

RQ2: What degree of non-determinism do we observe for LLMs when classifying flaky tests?

RQ3: To what degree do humans consider themselves capable of classifying flaky tests based on the test code?

Our results suggest that LLMs struggle to classify flaky tests solely on the basis of the test code. Furthermore, even with greedy decoding at temperature 0, the degree of non-determinism across repeated executions of our experiments complicates the deployment of such models in real-world scenarios for flakiness detection. Based on manual assessment, we conclude that test code alone is insufficient to detect certain types of flakiness. While there are issue types, such as the use of unordered collections, that may be apparent in the test code, we find that even humans require additional context information to identify more sophisticated flakiness issues. For example, flakiness may be caused by side effects of test utility functions whose functionality is not visible in the test code.

In summary, our work provides the following contributions:

1) Evaluation of three LLMs for flakiness classification on two common benchmarks.

*Also affiliated with SAP.

2640-7574/26/$31.00 ©2026 IEEE
DOI 10.1109/SANER67736.2026.00016

2) Discussion of implications for future work on flakiness classification.

The remainder of this document is structured as follows. Section II contains relevant information on the datasets used. We introduce relevant work in Section III before we pinpoint existing issues in one prior study in Section IV. We describe our methodology in Section V and present our results in Section VI-A. We discuss the results in Section VII and threats to validity in Section VIII, before concluding the paper in Section IX. Our code and the experimental results are available online as part of our supplementary material [19].

II. Datasets

We used two benchmark datasets for flakiness classification to evaluate LLMs for flakiness classification: the *International Dataset of Flaky Tests* (IDoFT) and *FlakeBench* [20], [21].

1. IDoFT: The IDoFT dataset arises from an open-source GitHub repository where contributors track flaky tests in open-source Java and Python projects [20], [22]. In this study, we utilized a subset of IDoFT that was previously used as a benchmark for binary flakiness classification [12], [17]. This subset contains 3813 samples, of which 587 (15%) were labeled non-flaky and 3226 (85%) were labeled flaky. Non-flaky samples represent flaky tests that have been fixed in a pull request with the status "Accepted". The samples in the dataset originate from 299 projects. In addition to a label indicating whether a test is flaky, Fatima et al. [5] have added category labels for the type of fix for each of the fixed tests. Labeling was performed using heuristics that Fatima et al. created based on a manual inspection of 100 samples. Table I provides an overview of the resulting fix labels and their prevalence in the given data.

2. FlakeBench: The FlakeBench dataset originates from 97 open-source repositories written in Java on GitHub. Rahman et al. [21] repeatedly executed tests in the respective repositories 100 times to identify flaky tests. Given the results from 100 repeated executions, they labeled a test flaky if it yielded at least one passing and one failing result. This labeling approach resulted in a dataset consisting of 8574 tests, of which 280 were labeled flaky and 8294 non-flaky.

III. Related Work

Prior work proposed various approaches to detect flaky tests based on different datasets [7], [9]–[18], [23]. In this paper, we focus on related work that used the test code to predict whether a test is flaky [7], [12]–[18], [21].

Pinto et al. [13] introduced the idea of using the test code for flakiness prediction based on the assumption that the test code of flaky tests exhibits syntactical patterns differentiating them from non-flaky tests. For example, flaky tests may be more likely to use certain words, such as "random" in identifier names. To test this assumption, they trained a random forest classifier on a bag-of-words representation of the test code. They evaluated their approach using the Matthews Correlation Coefficient (MCC), which ranges from -1 to 1, with 0 representing the average performance of random guessing and 1 representing a perfect classifier. With their approach, Pinto et al. achieved an MCC of 0.9 on the *DeFlaker* benchmark [24], indicating a high correlation between the predicted labels and the ground truth.

The original approach of Pinto et al. was replicated and evaluated multiple times [7], [15], [16]. Haben et al. evaluated the original approach on the same dataset with a different evaluation setup. They evaluated the original approach with a time-sensitive setup, in which the model was trained on code from older revisions and tested on newer revisions of the source code, resulting in a decrease in MCC of up to 21%. Camara et al. also investigated the performance of the approach of Pinto et al. [13] in a different evaluation setup, where the training data originated from software projects other than the test data. They observed a notable decrease in performance, achieving a prediction accuracy of only 0.48. Berndt et al. [7] evaluated Pinto et al.'s approach in an industrial context. They found that a random forest classifier achieved an F1 score of 0.95 on an internal flakiness benchmark, which was derived from dedicated test executions for flakiness investigations, but failed to generalize to data from the production system. They concluded that the lack of generalizability prevented the classifier from being deployed in practice.

More recent work has evaluated fine-tuning LLMs for flakiness detection [12], [17], [18]. Fatima et al. utilized a fine-tuned version of *CodeBERT*, a pre-trained language model with 125 million parameters, achieving an F1 score of 98% on the IDoFT dataset [18]. However, as pointed out in a follow-up study by Rahman et al. [21], their implementation suffered from data leakage, resulting in distorted results. The results without data leakage remain unknown. Rahman et al. applied quantization to the fine-tuned CodeBERT model resulting from Fatima et al.'s study to increase the efficiency during inference. They achieved an F1 score of up to 94% on IDoFT while reducing training time and RAM usage by 25% and 48%, respectively. More et al. [12] evaluated a few-shot learning approach in an intra-project scenario, where a model is trained and tested with data from a single software project. Thus, their approach reduced training time to 10% while maintaining similar classification performance on the IDoFT dataset.

In contrast to previous work on flaky test detection, we explore the potential of LLMs without any task-specific fine-tuning to classify tests as flaky. Unlike previous work, our approach does not require task-specific training data but relies on the capability of recent LLMs to generalize across tasks. We conjecture that LLMs without additional fine-tuning could be a solution to overcoming the generalization issues that were previously mentioned [15], [16].

IV. Issues of Prior LLM-based Flakiness Classifiers

As described in Section III, replications of previous work have already found that flakiness classifiers trained only on the test code do not generalize to new data from other projects [16] or future iterations of the same project [7], [15]. For example, in a study by Berndt et al., the authors noted that the lack

TABLE I: Summary of the fix categories for flaky tests as identified by Fatima et al. [5]. Note that one test may have multiple fix labels. Therefore, the sum of the support does not match the sample count of the dataset.

Fix type	Support	Description
Change assertion	175	Replacement of the employed assert function, e.g., `assertJsonStringEquals` instead of `assertEquals`
Change condition	109	Replacement of a condition within the assert statement, e.g., replacing `containsExactly` with `containsExactlyInAnyOrder`
Reset variable	98	Addition of a statement to reset the current state that was altered by the test, e.g., by calling a `clear` function
Change data structure	77	Replacement of the employed data structure, e.g. `LinkedHashMap` instead of `HashMap`
Reorder data	39	Addition of some sorting statement, e.g., calling `sort` on an array
Miscellaneous	37	Everything else
Reorder parameters	35	Change of parameter order in a function call
Change data format	30	Change of the format of some variable, e.g., from string to object
Call static method	9	Addition of a call to static method that encapsulates complex behavior, e.g., for test setup
String matching	5	Change matching of string-encoded representations, e.g., for comparing JSON objects with a string
Change time zone	3	Setting a specific time zone to avoid timing issues
Sleep	3	Addition of a call to the `sleep` function, e.g., to avoid race conditions
Handle timeout	2	Addition of a timeout value for proper timeout handling

of generalization was caused by an over-reliance on spurious features and a lack of diversity in the given datasets [7]. These existing replication studies used the original approach of Pinto et al. [13], which utilized a random forest classifier and a bag-of-words representation of the test code. However, we assume that newer studies using LLMs also suffer from spurious features and may therefore also struggle to generalize to new contexts.

To support this assumption, we analyzed the replication package of a recent study on flakiness classification by More et al. [25]. Similar to other studies on LLM-based flakiness classification, More et al. used the IDoFT dataset to benchmark their approach [17], [18]. They trained and evaluated their model in a per-project scenario, where a separate model was trained on 80% of the data for each project and tested on the remaining 20%.

A notable fraction of the projects in the IDoFT dataset is highly skewed towards flaky samples. For example, in the notebook for the `nifi` project, we found that the resulting classifier achieved an F1 score of 91.5% even though it classifies all tests as flaky [26]. In fact, we found that their approach yielded weighted F1-scores greater than 90% in 4 of 12 evaluated projects without correctly classifying any non-flaky samples [27], [27]–[29]. Therefore, we conclude that the classifier did not learn meaningful patterns to classify tests.

Based on an analysis of the remaining projects, we observed that the samples in a per-project scenario show low diversity. For example, for the two remaining projects with the highest support, `junit-quickcheck` and `spring-data-r2dbc`, almost all non-flaky samples shared the same piece of code fixing the flakiness. Figure 2a illustrates an example of the `junit-quickcheck` project, which had the highest support in the dataset. In this case, the addition of the line `Enums.iterations=0;` distinguishes the non-flaky version of the test from the flaky version. We observed that this statement holds for the majority of samples in this project. Thus, classifying only based on the line `iterations = 0` yields an F1 score of 96% when evaluated in the per-project scenario used in the study.

TABLE II: Landis et al.'s interpretation of Cohen's Kappa [31].

Values	Interpretation
≤ 0	Poor
$[0.01, 0.2]$	Slight
$[0.21, 0.4]$	Fair
$[0.41, 0.6]$	Moderate
$[0.61, 0.8]$	Substantial
$[0.81, 1]$	Almost perfect

Therefore, we conclude that their approach is likely to suffer from the same problems pointed out in previous studies on the weaknesses of flakiness classifiers based on the vocabulary of the test code. More and Jeremy reported a weighted F1 score of 95.1% across the projects in the IDoFT dataset [12]. However, based on the result in a Jupyter notebook in the replication package [30], we found that the actual weighted F1 score is 88.1%. Upon notification, the authors acknowledged this mistake in the paper and confirmed the reported F1 score in the notebooks.

V. Methodology

In this section, we describe our methodology for answering the research questions.

A. RQ1: Classification

For the first research question, we used three models, `GPT-4o`, `GPT-OSS-120b`, and `Qwen3-Coder-480b` [32]–[34]. We used all models with a temperature of 0 throughout the study to minimize the inherent non-determinism of the models [35]. We deployed `GPT-OSS-120b` and `Qwen3-Coder-480b` on a machine with 8 NVIDIA H200 GPUs using vLLM [36]. We evaluated four different settings for the prompt, as prompt engineering has been shown to improve the performance of LLMs for various tasks [37], [38]. We differentiated the three settings for our prompt based on the components that constitute it. Figure 1 illustrates our prompt template.

Task description: The task description x_{desc} describes the task to be performed by the LLM. For our case, we used the task *Classify the test as either flaky or not*.

Instructions for implicit reasoning: Based on OpenAI's prompt engineering guidelines [39], we added a set of instructions containing intermediate reasoning steps x_{reason} to approach the problem of binary flakiness classification. More specifically, we instructed the model to reflect on potential issues in the test code before providing the answer.

Demonstrations: Previous work has demonstrated that LLMs can learn from demonstrations within a given context through in-context learning [40]. That is, given a set of labeled samples in the context, an LLM can infer the underlying pattern and apply it to new inputs without any parameter updates. To benefit from this capability, we added $k = 6$ annotated samples of test code t to the prompt. We added three flaky ($y = 1$) and three non-flaky ($y = 0$) samples to avoid biasing the model towards one of the two classes, such that $x_{demo} = \{(t_1, y_{t_1}), ..., (t_6, y_{t_6})\}$. For each of the demonstrations in x_{demo}, we added the following three intermediate reasoning steps explaining the thought process to reach a conclusion on y given x_t:

- A semantic description of the test code.
- An elaboration on the critical lines related to flakiness.
- The conclusion on whether the test is considered flaky.

Based on these three components, we defined the three settings for our prompt that we evaluated in this study as follows:

1) $x_{zero} = \{x_{desc}\}$
2) $x_{zero-CoT} = \{x_{desc}; x_{reason}\}$
3) $x_{few-CoT} = \{x_{desc}; x_{reason}; x_{demo}\}$

Figure 1 illustrates the structure of x_{CoT}.

To compare our results with those of previous studies, we calculated the weighted precision, weighted recall, and the weighted F1 score as metrics [12]. As a baseline, we also calculated the results for classifying all tests as flaky and a random prediction with 50% chance of classifying a test as flaky. Furthermore, we added MCC as an evaluation metric to account for the skewed class distributions in the datasets.

B. RQ2: Non-Determinism

We compared two result vectors from an LLM obtained in two repeated experiments using greedy decoding with a temperature of 0. For example, classifying n tests as flaky (or not) results in a vector of length n, containing only 0 (non-flaky) and 1 (flaky). We measured the distance between two of such vectors using the normalized Hamming distance [41]. The Hamming distance H measures the distance between two vectors x and y of length n as follows:

$$H(x, y) = \frac{1}{n} \sum_{i=1}^{n} 1[x_i \neq y_i] \quad (1)$$

That is, intuitively, the Hamming distance measures the number of transitions from 0 to 1 required to transform one vector into the other.

C. RQ3: Human Judgement

In our third research question, we aimed to determine whether humans are capable of classifying tests as flaky solely based on the test code. To achieve this, we conducted a manual survey of $n = 50$ flaky examples from the IDoFT dataset, sampled from the subset provided by Fatima et al. [5] as described in Section II. Based on the question "How likely would a developer proficient in Java classify this test as flaky?", we labeled tests on a five-point Likert scale item ranging from 1 (Very likely) to 5 (Very unlikely). Furthermore, we included an open text field for the annotators to provide additional comments.

To create a common understanding between the two reviewers, we preceded the labeling with an alignment step. That is, we selected a preliminary set of 10 random samples and discussed the methodology to be used for labeling. Thus, the reviewers agreed on the following framework:

- The time box is five minutes per sample.

Exemplified x_{CoT}

Role: Expert Software Engineer.
Task: Classify the provided test as flaky or not.
Instructions:
1) Analyze the <CODE> segment, which contains an existing test.
2) Think about ways to improve the test or to fix existing issues with the test code.
3) Follow the reasoning style shown in the examples to identify potential sources of flakiness. Think step by step.
4) Classify the test as either flaky or not based on your thoughts on existing issues.
5) In your response, only include the following labels, refrain from using any natural language.
   0: the test is not flaky
   1: the test is flaky

Examples:
<EXAMPLE_CODE>
{example_code}
</EXAMPLE_CODE>
<THOUGHTS>
– {semantic_test_description}
– {flakiness_description}
– {conclusion}
</THOUGHTS>
<ANSWER>
{label}
</ANSWER>

[... additional examples]

<CODE>
{code}
</CODE>

Fig. 1: The structure of our x_{CoT} prompt template.

- Reviewers may look up only Java or test framework-specific knowledge if necessary (i.e., no project-specific knowledge).
- If the problem in the test code is not obvious at first sight, reviewers may look up the existing fix before labeling.

We quantified the inter-rater agreement between the two reviewers using Cohen's Kappa, as implemented by `scikit-learn` [42]. The values of Cohen's Kappa are within the interval $[-1, 1]$. Table II shows the interpretation of Cohen's Kappa values. We applied the weighted version of Cohen's Kappa, using quadratic weights, to account for the ordinal nature of the rating scale.

VI. Results

A. RQ1: Classification

We report the results of the classification across three iterations in Table III for *IDoFT* and Table IV for *FlakeBench*.

On the *IDoFT* dataset, $x_{few-CoT}$ achieved the best result for all models. While Qwen-Code achieved similar results for x_{zero} and $x_{zero-CoT}$, the two general-purpose models achieved higher results with $x_{few-CoT}$ by a large margin. These results suggest that both general-purpose models picked up relevant patterns from the examples in the context, thereby improving their overall performance. GPT-4o achieved an MCC of 0.12 with $x_{few-CoT}$, representing an improvement of a factor of 4 compared to other prompting techniques. GPT-OSS improved on the results of GPT-4o for all prompting techniques, achieving an MCC of 0.27 for $x_{few-CoT}$. However, an MCC of 0.27 still only indicates a weak correlation between the ground-truth labels and the model's predictions.

For the *FlakeBench* dataset, GPT-OSS also yielded the best results with every prompt. Comparing prompting techniques, x_{zero} achieved the best result on FlakeBench with an MCC of 0.17, in contrast to IDoFT. However, while $x_{few-CoT}$ improved the results of x_{zero} by a large margin on IDoFT, the two prompts resulted in similar results on the FlakeBench dataset ($MCC = 0.17$ vs. $MCC = 0.15$). $x_{zero-CoT}$ achieved the worst results on both datasets. Similar to the other two prompts, however, the differences in performance are less pronounced on the FlakeBench dataset. On FlakeBench, we observed a 0.03 decrease in performance for $x_{zero-CoT}$. On the IDoFT dataset, the largest margin between prompts is -0.30 ($x_{zero-CoT}$ vs. $x_{few-CoT}$).

We observed only minor differences in the performance metrics between different experiment runs. In most cases, there was no difference greater than 0.01 in MCC between the three repetitions. The largest range was between -0.03 and 0.01 for $x_{zero-CoT}$ on IDoFT.

> **Answer to RQ1**: We found that LLMs show a poor performance in classifying tests as flaky (or not) on both datasets. On *IDoFT*, we obtained the best result with few-shot prompting, achieving an MCC of 0.27. By F1 score, none of the approaches surpassed the *All flaky* baseline on IDoFT. On *FlakeBench*, the zero-shot prompt achieved the best result with an MCC of 0.17.

B. RQ2: Reliability

We report the normalized Hamming distances between three repetitions in Table V for *FlakeBench* and Table VI for *IDoFT*.

As shown in Table V and Table VI, $x_{zero-CoT}$ exhibited the highest non-determinism for GPT-4o and GPT-OSS on both datasets, resulting in normalized Hamming distances of up to 0.25. For GPT-4o, the difference between $x_{zero-CoT}$ and the results for the other two prompts was more pronounced, with $x_{zero-CoT}$ showing an increase in the normalized Hamming distance by at least a factor of 2. For GPT-OSS, the non-determinism increased by a factor of 1.07 for $x_{zero-CoT}$.

For Qwen-Code, $x_{few-CoT}$ resulted in the highest level of non-determinism on both datasets. Compared to the two GPT models, Qwen-Code exhibited a lower level of non-determinism overall. Furthermore, Qwen-Code was the only model that achieved equal results between two repetitions for x_{zero} and $x_{zero-CoT}$ on the FlakeBench dataset.

> **Answer to RQ2**: We found that non-determinism affects LLM-based flakiness classification. The non-determinism in our experiments ranged from 0% to 25% normalized Hamming distance. There were differences in non-determinism among models, datasets, and prompting techniques. Qwen-Code exhibited the lowest degree of non-determinism overall. GPT-OSS showed more non-determinism than GPT-4o.

C. RQ3: Manual Investigation

Figure 3 shows the labeling results from the two reviewers. Overall, the reviewers achieved a Kappa score of 0.57, indicating moderate agreement. Both reviewers produced an average rating close to the center of the scale (Reviewer 1: 3.08, Reviewer 2: 3.14). Rating 3 was used least frequently in only 8 out of 100 ratings, i.e., reviewers tended to make more decisive judgments towards the two ends of the scale. The most common rating was 2 (Likely) for 36 out of 100 ratings, followed by 5 (Very unlikely), which was used in 24 out of 100 ratings. As shown in Table VII, the reviewer ratings varied across the fix categories described in Table I. Reviewers considered changes to assertions or data structures in the test code easier to detect with an average rating of 2 (Likely). In contrast, the categories `reset_variable` and `reorder_parameters` received a rating of 4 (Unlikely).

Figure 2 illustrates two examples from the reset_variable and change_assertion. In Figure 2a, the test was fixed by resetting the `iterations` attribute of the `Enums` class. However,

TABLE III: Flakiness classification results on the IDoFT dataset for each prompt, separated by model. The numbers in bold represent the highest value in a column.

Approach	Model	Iter.	Prec.	Rec.	F1	MCC
x_{zero}	GPT-4o	1	0.88	0.14	0.24	0.03
		2	0.87	0.14	0.24	0.03
		3	0.88	0.14	0.24	0.04
	GPT-OSS	1	0.90	0.29	0.44	0.10
		2	0.91	0.30	0.43	0.12
		3	0.91	0.30	0.45	0.12
	Qwen-Code	1	0.91	0.26	0.41	0.11
		2	0.91	0.26	0.41	0.10
		3	0.91	0.26	0.41	0.11
$x_{zero-CoT}$	GPT-4o	1	0.84	0.78	0.81	0.00
		2	0.84	0.79	0.81	-0.03
		3	0.85	0.79	0.82	0.01
	GPT-OSS	1	0.87	0.58	0.70	0.06
		2	0.86	0.58	0.69	0.05
		3	0.86	0.56	0.68	0.05
	Qwen-Code	1	0.85	0.94	0.89	0.05
		2	0.85	0.94	0.89	0.05
		3	0.85	0.94	0.89	0.05
$x_{few-CoT}$	GPT-4o	1	0.87	0.83	0.84	0.12
		2	0.87	0.82	0.84	0.11
		3	0.87	0.82	0.84	0.12
	GPT-OSS	1	**0.91**	**0.75**	0.82	**0.27**
		2	0.91	0.75	0.82	0.26
		3	0.91	0.76	0.83	0.26
	Qwen-Code	1	0.89	0.49	0.63	0.11
		2	0.89	0.49	0.63	0.12
		3	0.89	0.52	0.65	0.12
All flaky			0.85	1	**0.92**	0.00
Random			0.84	0.51	0.64	0.00

TABLE IV: Flakiness classification results on the FlakeBench dataset for each prompt, separated by model. The numbers in bold represent the highest value in a column.

Approach	Model	Iter.	Prec.	Rec.	F1	MCC
x_{zero}	GPT-4o	1	0.06	0.42	0.11	0.09
		2	0.07	0.44	0.12	0.11
		3	0.07	0.44	0.12	0.10
	GPT-OSS	1	**0.10**	0.53	0.17	**0.17**
		2	0.10	0.53	0.17	0.17
		3	0.09	0.50	0.16	0.16
	Qwen-Code	1	0.08	0.57	0.15	0.16
		2	0.08	0.57	0.15	0.16
		3	0.08	0.57	0.15	0.16
$x_{zero-CoT}$	GPT-4o	1	0.05	0.7	0.09	0.09
		2	0.05	0.67	0.09	0.08
		3	0.05	0.70	0.09	0.08
	GPT-OSS	1	0.06	0.79	0.11	0.14
		2	0.06	0.79	0.11	0.14
		3	0.06	0.82	0.11	0.14
	Qwen-Code	1	0.04	0.95	0.08	0.09
		2	0.04	0.95	0.08	0.09
		3	0.04	0.95	0.08	0.09
$x_{few-CoT}$	GPT-4o	1	0.04	0.67	0.07	0.04
		2	0.04	0.70	0.07	0.05
		3	0.04	0.70	0.08	0.05
	GPT-OSS	1	0.06	0.88	0.11	0.15
		2	0.06	0.88	0.11	0.15
		3	0.06	0.88	0.11	0.15
	Qwen-Code	1	0.09	**1**	**0.17**	0.06
		2	0.10	1	0.18	0.06
		3	0.09	1	0.17	0.06
All flaky			0.03	1	0.06	0.00
Random			0.03	0.49	0.05	0.00

the existence of this attribute was not apparent from the test code. Furthermore, the increase of the variable happens as a side-effect of the call of `defaultPropertyTrialCount`, which was not defined in the test code either. Therefore, reviewers considered it unlikely that the test could be classified as flaky solely based on an understanding of the test code and without additional context. In contrast, in Figure 2b, the flakiness was caused by a mistakenly assumed order on the result of a stringified JSON representation. Since stringifying JSON does not guarantee an ordered representation of the key-value pairs, the test code was sufficient to identify and fix the issue. Both reviewers agreed that cases similar to these two example cases were common in the annotated test set. This suggests that some flakiness issues may be easier to detect, while others require additional context for clearance.

Answer to RQ3: Whether humans consider themselves capable of identifying a test as flaky based on the test code depends on the type of flakiness. While some types of flakiness were apparent in the test code, e.g., the reliance on the order of an unordered collection in the test, other types, such as problems caused by side effects of the tested functionality or test utility functions, were not visible in the test code.

TABLE V: Normalized Hamming distances between iterations of the same prompt for the FlakeBench dataset.

Model	Prompt	$H(y_1, y_2)$	$H(y_2, y_3)$	$H(y_1, y_3)$
GPT-4o	x_{zero}	0.08 (720)	0.08 (667)	0.08 (667)
	$x_{zero-CoT}$	**0.16** (1401)	**0.25** (2134)	**0.25** (2307)
	$x_{few-CoT}$	0.06 (518)	0.06 (477)	0.07 (605)
GPT-OSS	x_{zero}	0.06 (502)	0.07 (596)	0.06 (556)
	$x_{zero-CoT}$	**0.13** (1130)	**0.13 (1192)**	**0.13** (1150)
	$x_{few-CoT}$	0.12 (992)	0.12 (999)	0.11 (971)
Qwen-Code	x_{zero}	0.01 (120)	0.00 (0)	0.01 (120)
	$x_{zero-CoT}$	0.00 (5)	0.00 (0)	0.00 (5)
	$x_{few-CoT}$	**0.13** (1104)	0.00 (1)	**0.13** (1103)

TABLE VI: Normalized Hamming distances between iterations of the same prompt for the IDoFT dataset.

Model	Prompt	$H(y_1, y_2)$	$H(y_2, y_3)$	$H(y_1, y_3)$
GPT-4o	x_{zero}	0.01 (52)	0.01 (48)	0.01 (52)
	$x_{zero-CoT}$	**0.09 (344)**	**0.09 (362)**	**0.09 (366)**
	$x_{few-CoT}$	0.02 (96)	0.02 (83)	0.02 (87)
GPT-OSS	x_{zero}	0.12 (473)	0.12 (463)	0.12 (464)
	$x_{zero-CoT}$	**0.15 (583)**	**0.15 (580)**	**0.15 (575)**
	$x_{few-CoT}$	0.13 (496)	0.13 (524)	0.12 (472)
Qwen-Code	x_{zero}	0.01 (21)	0.00 (19)	0.01 (20)
	$x_{zero-CoT}$	0.00 (2)	0.00 (2)	0.00 (2)
	$x_{few-CoT}$	**0.01 (36)**	**0.01 (36)**	**0.01 (36)**

```
public void enums() throws Exception {
  assertThat(testResult(Enums.class),isSuccessful());
  assertEquals(defaultPropertyTrialCount(),Enums.iterations);
  assertEquals(EnumSet.of(HALF_UP,HALF_EVEN),new HashSet<>(Enums.values.subList(0,2)));
+ Enums.iterations=0;
}
```

(a) Side-effects can modify `Enums.iterations`, which is not apparent in the test. Reviewers considered a flaky classification unlikely.

```
public void TestBlikDetailsSerialization() throws JsonProcessingException {
  String expectedJson=""{...}"";
  BlikDetails blikDetails=new BlikDetails();
  blikDetails.setBlikCode(""blikCode"");
  PaymentsRequest paymentsRequest=createPaymentsCheckoutRequest();
  paymentsRequest.setPaymentMethod(blikDetails);
  String gson=GSON.toJson(paymentsRequest);
  assertEquals(expectedJson,gson);
  String jackson=OBJECT_MAPPER.writeValueAsString(paymentsRequest);
- assertEquals(expectedJson,jackson);
+ assertJsonStringEquals(expectedJson,jackson);
}
```

(b) JSON does not guarantee an order when stringified. Reviewers agreed that it is likely to detect the test as flaky.

Fig. 2: Examples of `reset_variable` and `change_assertion` (diff indicates fixed version of the corresponding test).

TABLE VII: Average ratings aggregated by fix category.

Fix category	Support	Mean
change_assertion	14	2.2
change_condition	8	2.9
misc	6	4.4
change_data_structure	6	2.7
reset_variable	5	3.9
reorder_parameters	3	4.2
reorder_data	3	3.0
change_assertion,change_condition	2	3.5
reset_variable,handle_exceptions	2	4.0
reorder_data,change_data_structure	1	3.0
reorder_data,change_assertion,change_condition	1	2.0
sleep	1	5.0
change_data_format	1	1.5

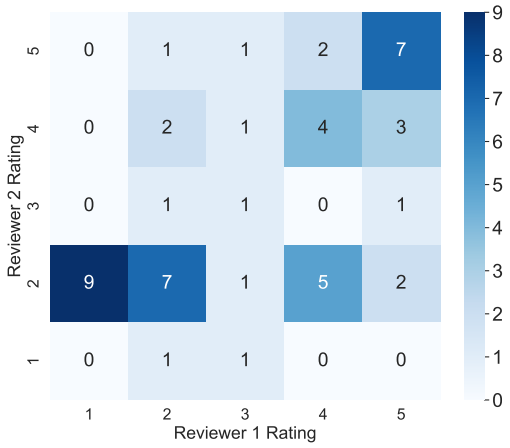

Fig. 3: Heatmap showing the results of the survey from two reviewers. Answers to the question "How likely would a developer proficient in Java classify this test as flaky?" range from 1 (Very likely) to 5 (Very unlikely).

VII. Discussion

In this section, we discuss our empirical results.

On the flakiness classification. We found that LLMs without fine-tuning struggle to classify tests as flaky solely based on the test code. Achieving an MCC of 0.27 in the best case, we argue that the performance is far below the requirements for deploying such a model in a production scenario. This is further aggravated by the exhibited non-determinism of the models when asked to classify a test. With the aim of deploying a flakiness detection model in practice in mind, it is crucial that the model reliably detects flaky tests. Based on their experience with the deployment of other AI systems in development processes, practitioners at SAP noted that developers quickly lose trust in a system that delivers inconsistent results for equal inputs [43]. We envision two possible future avenues to solve these problems. Firstly, to improve the models' predictive performance, future work could experiment with additional context, such as the code under test, documentation, or test helper functions. Based on preliminary manual experiments with examples from the IDoFT dataset, we found that presenting the model with the *relevant* context, which contains all necessary information, can improve the model's predictions. We conjecture that agentic systems or retrieval-augmented generation (RAG) approaches that dynamically fetch relevant information may yield better

results than those observed in our study. Secondly, to mitigate the negative effects of the model's non-determinism, future work could experiment with approaches such as self-consistency, ensembles, or combinations of the two [44].

On the labeling results. Based on our manual labeling of flaky tests in the IDoFT dataset, we found that classifying tests as flaky only based on the given test code may be infeasible for certain types of flakiness. Given the fix labels provided by previous work [5], we observed differences in the feasibility of classifying a test as flaky for tests with different fix labels. More specifically, while it may be possible to classify a test as flaky based on the existence of trivial test smells, such as the use of unordered collections or sleep statements in the test code, identifying the absence of proper setup or teardown requires further information about the code being tested.

This finding is also in line with previous research on fixing flaky tests [45], [46]. When using GPT-4 to fix flaky tests only given the test code, Chen et al. [45] found that performance varies between different types of flakiness. Although their approach successfully repaired 58% of flaky tests, whose flakiness was caused by implementation errors in the test code, the authors pointed out that GPT-4 was not capable of repairing other types of flakiness in their study.

Further research is required to identify the differences between test-only flakiness issues and more complex issues that occur in other parts of the source code. Based on our analysis of the IDoFT dataset, we conjecture that these different types of flakiness may also have varying impacts on testing practices. Although test-only issues can be easily identified and fixed by developers, more sophisticated issues can be challenging to debug and fix, as they often rely on more complex program states [47].

VIII. Threats to Validity

A. Internal Validity

In this section, we describe possible alternative explanations for our results [48].

On the LLM results. We examined a limited set of prompting techniques and LLMs. Other models or prompts might yield different results. Furthermore, the non-determinism of LLMs could have affected the results of our study. To mitigate this threat, we used greedy decoding and repeated our experiments three times [49]. Furthermore, we added more depth to the study by manually investigating 50 samples of the dataset. The results of our manual investigation align with the findings of our experiments using LLMs.

On the manual investigation. We manually examined 50 samples of the IDoFT dataset. We randomly sampled these 50 samples. Thus, the resulting sample may have suffered from sampling bias, which might lead to an overestimation of the proportion of tests in which flakiness is not apparent in the test code.

B. External Validity

We describe the extent to which our results generalize to other contexts [50].

On the employed datasets. Since we only used two datasets with a limited number of flaky tests written only in Java, our results may not generalize to all types of tests. However, we note that the results align with our experience in real-world CI/CD pipelines. For example, the ticket system at SAP offers a dedicated tag to flag test-only issues, where the fault lies within the test code. However, not all flakiness-related issues are tagged as test-only. In fact, differentiating between test-only issues and more complex flakiness issues is a common challenge.

C. Construct Validity

We describe the degree to which our metrics measure the intended properties [51].

On the evaluation metrics for the flakiness classification. Similar to prior studies on flakiness classification, we report the precision, recall, and F1 score for our prediction results [12], [17], [18]. Since the class distributions of the two datasets in this study are opposed, this results in measures that are hardly comparable. However, to mitigate this problem, we added the MCC, a common metric in the context of binary classification tasks, to enable a comparison of the results on the two datasets [52].

IX. Conclusions

We evaluated three LLMs without additional fine-tuning for flakiness classification, given only the test code. Based on evaluations across two benchmark datasets, our results show that LLMs struggle to correctly classify flaky tests. Our manual analysis of 50 samples suggests that this is due to the absence of necessary information in the test code for correct classification. This finding contradicts previously reported results from fine-tuning approaches, which suggested that LLM-based flakiness classification based solely on test code is promising.

While related work has already identified issues in a fine-tuning study, our examination of the replication package for another related study suggests that the relatively high reported scores are due to two reasons. On the one hand, a skewed evaluation setup in which simple baseline approaches already yield high scores. Second, we expect the high scores in the reported metrics to be due to overfitting to overly simple characteristics that differentiate flaky tests from non-flaky tests in the given dataset.

Furthermore, we evaluated the non-determinism of LLMs for flakiness classification. Our results indicate that the non-determinism varies between models, prompts, and datasets. We argue that the exhibited amount of non-determinism is problematic for deploying such models in practice.

Our findings can motivate future research evaluating general-purpose LLMs with additional sources of information beyond the test code. For this, we mainly see potential in the use of agentic systems or RAG, inspired by previous work on program repair [53], [54].

REFERENCES

[1] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.

[2] A. Berndt, T. Bach, and S. Baltes, "Do Test and Environmental Complexity Increase Flakiness? An Empirical Study of SAP HANA," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 572–581.

[3] 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*, 2019, pp. 830–840.

[4] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.

[5] S. Fatima, H. Hemmati, and L. Briand, "Flakyfix: Using Large Language Models for Predicting Flaky Test Fix Categories and Test Code Repair," *IEEE Transactions on Software Engineering*, 2024.

[6] X. Liu, Z. Song, W. Fang, W. Yang, and W. Wang, "WEFix: Intelligent Automatic Generation of Explicit Waits for Efficient Web End-to-End Flaky Tests," in *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, T. Chua, C. Ngo, R. Kumar, H. W. Lauw, and R. K. Lee, Eds. ACM, 2024, pp. 3043–3052. [Online]. Available: https://doi.org/10.1145/3589334.3645628

[7] A. Berndt, Z. Nochta, and T. Bach, "The Vocabulary of Flaky Tests in the Context of SAP HANA," in *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2023, pp. 1–9.

[8] B. Magill and P. McMinn, "deflake.rs: Detect Flaky Tests in Rust Projects using Execution Data," *J. Open Source Softw.*, vol. 10, no. 113, p. 8757, 2025. [Online]. Available: https://doi.org/10.21105/joss.08757

[9] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, and Y. Le Traon, "The Importance of Accounting for Execution Failures when Predicting Test Flakiness," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1979–1989.

[10] M. Hoang and A. Berding, "Presubmit Rescue: Automatically Ignoring FlakyTest Executions," in *Proceedings of the 1st International Workshop on Flaky Tests*, 2024, pp. 1–2.

[11] J. Lampel, S. Just, S. Apel, and A. Zeller, "When Life Gives you Oranges: Detecting and Diagnosing Intermittent Job Failures at Mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1381–1392.

[12] R. More and J. S. Bradbury, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 349–359. [Online]. Available: https://doi.org/10.1109/ICST62969.2025.10988995

[13] 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.

[14] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you Neighbor: Fast Static Prediction of Test Flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[15] 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," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 219–229.

[16] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio, "What is the Vocabulary of Flaky Tests? An Extended Replication," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 444–454.

[17] S. Rahman, A. Baz, S. Misailovic, and A. Shi, "Quantizing Large-Language Models for Predicting Flaky Tests," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*. IEEE, 2024, pp. 93–104. [Online]. Available: https://doi.org/10.1109/ICST60714.2024.00018

[18] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A Black-box, Language Model-based Predictor for Flaky Tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1912–1927, 2022.

[19] A. Berndt, V. Bekmyradov, R. Gemulla, M. Kessel, T. Bach, and S. Baltes, "Can We Classify Flaky Tests Using Only Test Code? An Empirical Study using LLMs," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.18311270

[20] W. Lam, "International Dataset of Flaky Tests (IDoFT)," 2020. [Online]. Available: http://mir.cs.illinois.edu/flakytests

[21] S. Rahman, S. Dutta, and A. Shi, "Understanding and Improving Flaky Test Classification," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3763098

[22] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST 2019: 12th IEEE International Conference on Software Testing, Verification and Validation*, Xi'an, China, April 2019, pp. 312–322.

[23] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms using Association Rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 39–48.

[24] 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 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 433–444. [Online]. Available: https://doi.org/10.1145/3180155.3180164

[25] R. More and J. S. Bradbury, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/tree/f7b7b270928dc2a1b7c8c6ce23730869db9d6bc7

[26] ——, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/blob/main/src/IDoFT%20dataset%20code/FlakyXbert-IDoFT_binary_projectwise_nifi.ipynb

[27] ——, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/blob/main/src/IDoFT%20dataset%20code/FlakyXbert-IDoFT_binary_projectwise_hadoop.ipynb

[28] ——, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/blob/main/src/IDoFT%20dataset%20code/FlakyXbert-IDoFT_binary_projectwise_fastjson.ipynb

[29] ——, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/blob/main/src/IDoFT%20dataset%20code/FlakyXbert-IDoFT_binary_projectwise_admiral.ipynb

[30] ——, "An Analysis of LLM Fine-Tuning and Few-Shot Learning for Flaky Test Detection and Classification - Replication Package," 2025. [Online]. Available: https://github.com/seer-lab/FlakyXbert/blob/main/src/IDoFT%20dataset%20code/calculate.ipynb

[31] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," *biometrics*, pp. 159–174, 1977.

[32] OpenAI, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[33] OpenAI, "gpt-oss-120b & gpt-oss-20b Model Card," 2025. [Online]. Available: https://arxiv.org/abs/2508.10925

[34] Q. Team, "Qwen3 Technical Report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[35] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An Empirical Study of the Non-determinism of Chatgpt in Code Generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–28, 2025.

[36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language Models are Few-shot Learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[38] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-shot Reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[39] OpenAI, "Prompt Engineering," 2025. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering/strategy-give-models-time-to-think#tactic-use-inner-monologue-or-a-sequence-of-queries-to-hide-the-model-s-reasoning-process

[40] A. Lampinen, I. Dasgupta, S. Chan, K. Mathewson, M. Tessler, A. Creswell, J. McClelland, J. Wang, and F. Hill, "Can Language Models Learn From Explanations in Context?" in *Findings of the Association for Computational Linguistics: EMNLP 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 537–563. [Online]. Available: https://aclanthology.org/2022.findings-emnlp.38/

[41] Y. Sun, H. Wang, D. Li, G. Wang, and H. Zhang, "The Emperor's New Clothes in Benchmarking? A Rigorous Examination of Mitigation Strategies for LLM Benchmark Data Contamination," *arXiv preprint arXiv:2503.16402*, 2025.

[42] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[43] S. Baltes, T. Speith, B. Chiteri, S. Mohsenimofidi, S. Chakraborty, and D. Buschek, "On the Need to Rethink Trust in AI Assistants for Software Development: A Critical Review," *arXiv preprint arXiv:2504.12461*, 2026.

[44] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-Consistency Improves Chain of Thought Reasoning in Language Models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=1PL1NIMMrw

[45] Y. Chen, "Flakiness Repair in the Era of Large Language Models," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 441–443.

[46] Y. Chen and R. Jabbarvand, "Neurosymbolic Repair of Test Flakiness," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1402–1414. [Online]. Available: https://doi.org/10.1145/3650212.3680369

[47] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding Reproducibility and Characteristics of Flaky Tests through Test Reruns in Java Projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 403–413.

[48] M. B. Brewer and W. D. Crano, "Research Design and Issues of Validity," *Handbook of research methods in social and personality psychology*, pp. 3–16, 2000.

[49] S. Baltes, F. Angermeir, C. Arora, M. M. Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, D. Fucci, M. Kalinowski, S. Lambiase, D. Russo, M. Lungu, L. Prechelt, P. Ralph, R. van Tonder, C. Treude, and S. Wagner, "Guidelines for Empirical Studies in Software Engineering involving Large Language Models," 2025. [Online]. Available: https://arxiv.org/abs/2508.15503

[50] S. Baltes and P. Ralph, "Sampling in Software Engineering Research: A Critical Review and Guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022.

[51] P. Ralph and E. Tempero, "Construct Validity in Software Engineering Research and Software Metrics," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. ACM, 2018, pp. 13–23.

[52] D. Chicco and G. Jurman, "The Advantages of the Matthews Correlation Coefficient (MCC) over F1 score and Accuracy in Binary Classification Evaluation," *BMC genomics*, vol. 21, no. 1, p. 6, 2020.

[53] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair," *arXiv preprint arXiv:2403.17134*, 2024.

[54] P. Rondon, R. Wei, J. Cambronero, J. Cito, A. Sun, S. Sanyam, M. Tufano, and S. Chandra, "Evaluating agent-based program repair at google," *arXiv preprint arXiv:2501.07531*, 2025.