Toward Static Test Flakiness Prediction: A Feasibility Study

Valeria Pontillo
SeSa Lab — Department of Computer Science, University of Salerno
Fisciano, Italy
vpontillo@unisa.it

Fabio Palomba
SeSa Lab — Department of Computer Science, University of Salerno
Fisciano, Italy
fpalomba@unisa.it

Filomena Ferrucci
SeSa Lab — Department of Computer Science, University of Salerno
Fisciano, Italy
fferrucci@unisa.it

ABSTRACT

Flaky tests are tests that exhibit both a passing and failing behavior when run against the same code. While researchers attempted to define approaches for detecting and addressing test flakiness, most of them suffer from scalability issues. This limitation has been recently targeted through machine learning solutions that could predict the flakiness of tests using a set of both static and dynamic metrics that would avoid the re-execution of tests. Recognizing the effort spent so far, this paper poses the first steps toward an orthogonal view of the problem, namely the classification of flaky tests using *only* statically computable software metrics. We propose a *feasibility study* on 72 projects of the iDFlakies dataset, and investigate the differences between flaky and non-flaky tests in terms of 25 test and production code metrics and smells. First, we statistically assess those differences. Second, we build a logistic regression model to verify if the differences observed are still significant when the metrics are considered together. The results show a relation between test flakiness and a number of test and production code factors, indicating the possibility to build classification approaches that exploit those factors to predict test flakiness.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**.

KEYWORDS

Flaky Tests; Software Quality Evaluation; Empirical Studies.

ACM Reference Format:
Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. 2021. Toward Static Test Flakiness Prediction: A Feasibility Study. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution (MaLTESQuE '21), August 23, 2021, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3472674.3473981

1 INTRODUCTION

Regression testing consists of verifying newly committed code changes for the presence of software faults [31]. Developers rely on test cases to decide on whether to merge pull requests or even deploy the entire system [11]. Perhaps more importantly, developer's productivity is partially dependent on the outcome of test cases [4, 22]: this is mainly due to their ability to identify real faults in a timely and reliable fashion [30].

Unfortunately, tests can be defective as well and, sometimes, they can be affected by the so-called *flakiness* [19]: this happens when a test exhibits both a passing and failing behavior when run against the same code, being therefore unreliable and producing a non-deterministic outcome. According to the research literature on the matter, flaky tests (1) may hide real defects and be hard to reproduce because of their non-determinism [19]; (2) increase testing costs, as developers invest time debugging failures that are not real [15]; (3) can reduce the overall developer's confidence on test cases, potentially leading to neglect real defects [7].

The consequences of test flakiness have been made more and more popular by practitioners and companies worldwide (e.g., [8, 22]), who all called for automated mechanisms to detect them.

The software engineering research community has been contributing to the body of knowledge through empirical investigations aiming at eliciting the causes of flakiness [7, 17–19, 21] as well as with the definition of techniques for detecting and addressing them [2, 6, 34, 38]. Despite the promising results achieved so far, most of the identification techniques require test cases to be re-run multiple times: as an example, the most well-known approach is called ReRun and consists of executing the same test N times, with N being a variable that goes from dozens to hundreds of executions. As the reader may understand, the poor scalability of ReRun makes it often unusable in practice; in addition, there is no guarantee to discover the flakiness over the N runs.

To overcome this limitation, researchers devised some alternatives, like DeFlaker [2], that work at commit-level and rely on the differential code coverage extracted from the analysis of a test execution from a commit to another. In a complementary manner, the use of machine learning approaches has been proposed. Pinto et al. [32] and further replications [3, 13] exploited the test code dictionary to discriminate the presence of potential flakiness. More recently, Alshammari et al. [1] devised a supervised learning model that, using a mixture of code and coverage metrics, can predict flaky tests with an accuracy up to 86%.

Inspired by these previous papers on flakiness prediction, in this work we aim at pursuing a further step forward. Rather than predicting flaky tests with a combination of metrics that include possibly costly dynamic features (e.g., code coverage), we aim at studying the feasibility of devising a set of fully statically computable features to be used for test flakiness prediction. Our work has therefore a *feature engineering* connotation and aims at posing the first steps toward our goal. We compute 25 factors related

MaLTESQuE '21, August 23, 2021, Athens, Greece

ACM ISBN 978-1-4503-8625-8/21/08…$15.00
https://doi.org/10.1145/3472674.3473981

to production and test code on 72 open-source projects coming from the iDFlakies dataset of annotated flaky tests [16]. In the first place, we verify the distribution of each individual factor in the sets of flaky and non-flaky tests, with the aim of seeking the metrics that statistically differ and that might possibly characterize flaky tests. Afterwards, we build a logistic regression model to evaluate whether the differences observed in the first phase are still significant when the factors are combined together.

Our preliminary results reveal a relation between test flakiness and a number of test and production code factors. For instance, we find the complexity of production code to be statistically different in the two sets of tests, being therefore a candidate metric for a machine learner approach. Our findings showcase the feasibility of building a fully static approach for flaky test prediction. All data, scripts, and additional analyses conducted in the experiments are available in our online appendix [33].

Table 1: List of metrics used as independent variables.

Name	Description
Production and Test Code Metrics	
CBO	Coupling Between Object, i.e., the number of dependencies a class has with other classes [5].
Halstead Length	The total number of operator occurrences and the total number of operand occurrences.
Halstead Vocabulary	The total number of distinct operators and operands in a function.
Halstead Volume	Proportional to program size, represents the size, in bits, of space necessary for storing the program.
LOC	Lines of Code, counting both source and comment lines.
LCOM2	Lack of Cohesion of Methods version 2, i.e., the percentage of methods that do not access a specific attribute averaged over all attributes in the class.
LCOM5	Lack of Cohesion of Methods version 5, i.e., the density of accesses to attributes by methods.
McCabe	Weighted Methods per Class, i.e., the sum of the complexities (i.e., McCabe's Cyclomatic Complexity) of all the methods in a class [20].
MPC	Message Passing Coupling, measures the numbers of messages passing among objects of the class.
RFC	Response For a Class, i.e., the number of methods (including inherited ones) that can potentially be called by other classes [5].
TLOC	Number of lines of code of the Test Suite.
WMC	Weighted Methods per Class, i.e., the sum of the complexities (i.e., McCabe's Cyclomatic Complexity) of all the methods in a class [5].
Code Smells	
Class Data Should Be Private	When a class exposes its attributes, violating the information hiding principle.
Complex Class	When a class has a high cyclomatic complexity.
Functional Decomposition	When in a class inheritance and polymorphism are poorly used.
God Class	When a class has huge dimension and implementing different responsibilities.
Spaghetti Code	When a class has no structure and declares long method without parameters.
Test Smells	
Assertion Density	Percentage of assertion statements in the test code
Assertion Roulette	When a test method has multiple non-documented assertions.
Conditional Test Logic	Conditional code within a test method negatively impacts the ease of comprehension by developers.
Eager Test	When a test method invokes several methods of the production object.
Fire and Forget	A test that is at risk of exiting prematurely because it does not properly wait for the results of external calls.
Mystery Guest	When a test method utilizes external resources (e.g. files, database, etc.).
Resource Optimism	When a test method makes an optimistic assumption that the external resource (e.g., File), utilized by the test method, exists.
Sensitive Equal.	When the toString method is used within a test method.

2 RESEARCH QUESTIONS AND VARIABLES

The *goal* of the study was to investigate how static test and production quality metrics are related to test flakiness, with the *purpose* of assessing the feasibility of a flaky test prediction model solely based on statically computable features. The *perspective* is of researchers and practitioners: the former are interested in understanding the capabilities of code and test-related metrics when it comes to the identification of flaky tests; the latter are interested in evaluating which are the features more connected to flakiness and that, therefore, should be kept under control when evolving source code.

As further explained later in Section 2.3, our focus was on both test and production code metrics and smells. While the research community has identified test-related aspects as those primarily connected to the potential flakiness of test code [19], we considered production code metrics on the basis of the findings reported in a recent work by Eck et al. [7]: in a non-negligible number of cases, the root-cause of test flakiness might be due to errors done in the production code (e.g., when managing concurrency [7]). To the best of our knowledge, this is the first research work investigating the impact of production code quality factors on flaky tests.

This reasoning let us define our $\mathbf{RQ}_1$: we started by analyzing how the above mentioned metrics correlate to test flakiness. We focused on their individual effect by statistically comparing how their values differ in the sets of flaky and non-flaky tests. We asked:

> **$\mathbf{RQ}_1$.** *What are the individual effects of production and test code quality metrics on the prediction of flaky tests?*

While the results to the first research question might already provide insights into the relations between static metrics and test flakiness, we performed an additional step with the aim of verifying whether the differences observed in $\mathbf{RQ}_1$ were still statistically significant when the considered metrics were combined: as shown in literature [29], this step is required to establish unbiased conclusions on the capabilities of metrics for predictive models:

> **$\mathbf{RQ}_2$.** *What are the combined effects of production and test code quality metrics on the prediction of flaky tests?*

2.1 Context Selection

The *context* of the study consisted of 72 open-source software projects accounting for a total of 51,549 test cases, of which at least one affected by flakiness. More specifically, we relied on the publicly available iDFlakies dataset produced by Lam et al. [16]. It provides a collection of 423 flaky tests, identified by re-running them multiple times in different orders. The projects in the dataset are all available on Github and have high diversity: for instance, they are developed by 66 different communities, six projects belong to the Apache Software Foundation and have a size ranging from 638 to 1.6 million lines of code. More detailed statistics on those projects are available on the dataset website.[1]

The selection of this dataset was driven by its availability as well as by the large amount of diverse projects it contains.

2.2 Dependent Variable

The dependent variable of our study is the test flakiness, as reported in the iDFlakies dataset [16]. In particular, test cases are either labeled as *"flaky"* or *"non-flaky"*. As such, our statistical exercise will consider a binary dependent variable.

2.3 Independent Variables

The ultimate goal of our work was to verify the extent to which statically computable metrics can be adopted to predict test flakiness. In the context of this feasibility study, we considered a total of 25

[1] The iDFlakies dataset: https://sites.google.com/view/flakytestdataset/home.

factors along three dimensions i.e., *production and test code metrics*, *code smells*, and *test smells*. Table 1 reports name and description of the considered metrics, while the rationale and motivations for selecting them are discussed in the following.

Production and test code metrics. This set is composed of ten factors measuring size and complexity of both production and test code. Some of these metrics belong to the Object-Oriented metric suite proposed by Chidamber and Kemerer [5], e.g., coupling between object classes (CBO), while other metrics come from other catalogues, e.g., the McCabe cyclomatic complexity [20] or the Halstead's metrics [24]. The rationale behind the selection of these metrics was driven by our willingness to verify whether large and/or complex code might have an impact on the likelihood to observe a flaky behavior of the test case.

Code smells. These indicate the presence of sub-optimal solutions to the development of source code [9] that might contribute to the increase of technical debt [27]. It is reasonable to believe that writing tests for smelly code may be harder and might possibly lead them to be less effective—this was somehow showed by Grano et al. [12]. Hence, we run DECOR [23], a state-of-the-art code smell detector, to count the number of instances of five code smell types having different characteristics and targeting well-known design issues, i.e., *Class Data Should Be Private*, *Complex Class*, *Functional Decomposition*, *God Class*, and *Spaghetti Code* [9].

Test smells. Similarly to code smells, these are defined as bad programming practices in unit test code [36]. As originally defined, test smells may indeed reveal the presence of issues that induce test flakiness [36]: hence, we run a state-of-the-art test smell detector named VITRUM [28] to verify whether test smells have an impact on flakiness. The detector identifies seven test smell types, i.e., *Assertion Roulette*, *Conditional Test Logic*, *Eager Test*, *Fire and Forget*, *Mystery Guest*, *Resource Optimism*, and *Sensitive Equality*.

When computing metrics and smells on production code, we had to link test cases to their correspondent production code. In this respect, we used a pattern matching approach based on naming conventions and already used in previous work (e.g., [12, 13, 29]).

Once we had computed the above-mentioned factors on the considered projects, we normalized the resulting values using the *min-max scaling*—this was needed because the metric values were on different scales, making any comparison difficult [14].

3 RQ$_1$. THE INDIVIDUAL EFFECTS OF METRICS ON TEST FLAKINESS

3.1 Research Methodology

We assessed if the independent variables were different in the set of flaky and non-flaky sets. We first showed boxplots depicting the distribution of the metrics and smells. Then, we computed the Mann-Whitney and Cliff's Delta tests to verify the statistical significance of the observed differences and their effect size.

3.2 Analysis of the Results

Figure 1 depicts the boxplots of the distributions of metrics and smells which exhibit some differences between the sets of flaky and non-flaky tests. The boxplot for the entire factors is reported in our replication package [33]. We can observe that some factors

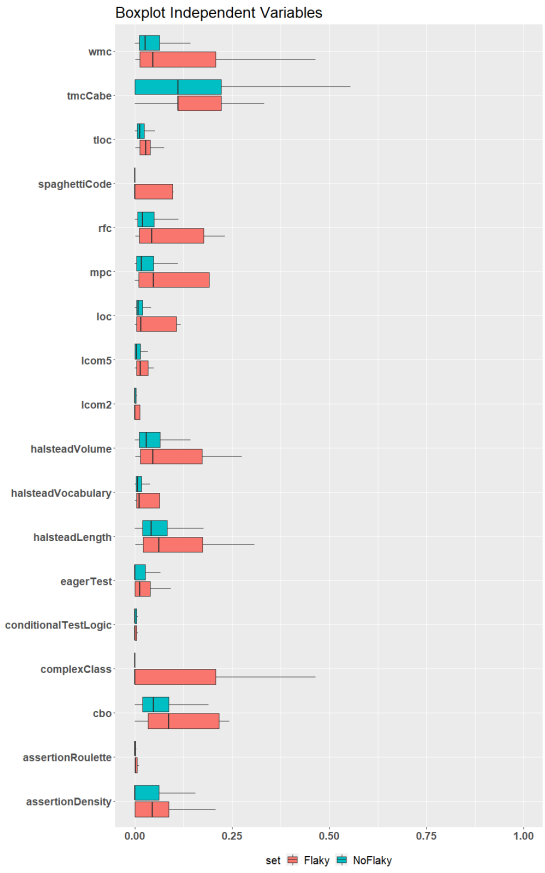

Figure 1: Results for RQ$_1$.

vary in the two sets: this is especially true when considering the production and test code metrics, for which the medians of flaky tests and corresponding production code are often higher than those of non-flaky tests. These results suggest that flaky tests have a different metric profile than other tests. Moreover, we observe a variation in terms of production code factors, which is not only in line with previous results [7], but also indicate a new, alternative dimension of the test flakiness neglected so far.

Perhaps more interesting, the boxplot analysis reveals that most of the metrics and smells that are different in the two sets characterize program complexity. In particular, we observe differences in terms of control flow graph-related metrics (e.g., production WMC and the McCabe metric computed on tests), complexity of the expressions used in the code (e.g., the Halstead's metrics), and code smells (e.g., Complex Class). This seems to suggest that the development of test cases is heavily impacted by complexity measures, possibly increasing the likelihood to induce flakiness. As such, we can envision future additional analyses on such a relation. As for the test-related factors, the higher median of assertion density in the flaky test set might be connected to the fact that having more assertions increases the chances to induce flakiness due to restrictive ranges in the values compared within `assert` statements [7]. Finally, the effects of test smells are only partially visible: indeed,

Table 2: Mann Whitney and Cliff's Delta Statistical Test Results. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant p-value and δ value are reported in bold-face.

Statistic Tests					
	p-value	δ		p-value	δ
CBO	**$3.47e^{-13}$**	**S**	Complex Class	**$< 2.2e^{-16}$**	**S**
Halstead Length	**$4.45e^{-05}$**	**S**	FD	0.19	N
Halstead Vocab.	**$9.97e^{-06}$**	**S**	God Class	0.19	N
Halstead Volume	**$2.65e^{-05}$**	**S**	Spaghetti Code	**$2.24e^{-15}$**	**S**
LOC	**$1.57e^{-07}$**	**S**	Assertion Density	**$8.01e^{-10}$**	**S**
LCOM2	**$7.48e^{-05}$**	**S**	Assertion Roulette	**$2.29e^{-12}$**	**S**
LCOM5	**$7.99e^{-07}$**	**S**	Cond. Test Logic	0.07	N
McCabe	**0.01**	N	Eager Test	**$7.66e^{-16}$**	**S**
MPC	**$4.22e^{-10}$**	**S**	Fire And Forget	**0.03**	N
RFC	**$4.02e^{-08}$**	**S**	Mystery Guest	0.52	N
TLOC	**$< 2.2e^{-16}$**	**M**	Resource Optimism	0.29	N
WMC	**$5.37e^{-06}$**	**S**	Sensitive Equality	0.71	N
CDSBP	**$< 2.2e^{-16}$**	N			

most of the test cases in the dataset were not affected by any of them, hence limiting the analysis and suggesting the need for larger investigations into the matter. Nonetheless, we observe the severity of the *Eager Test* smell as a metric that differs in two sets. This smell measures how focused a test is, namely whether it exercises more methods of the production code. Based on our results, we may conjecture that the lack of focus of tests does not allow them to properly set the environment needed to exercise the production code: as a consequence, their outcome may depend on the order of execution of test methods, i.e., the outcome may change if the environment is (not) set before calling the smelly test.

The results of the statistical tests are reported on Table 2 and confirm the discussion provided so far. Most of the metrics (19) present a ρ-value < 0.05, meaning that the differences between the distributions of flaky and non-flaky tests are statically significant. These differences have, however, a small effect size in 15 cases.

Key findings of RQ_1.

The metric profile of flaky tests is different from the one of non-flaky tests. One the key factors distinguishing them is program complexity. We also discovered a set of production code metrics that might potentially affect test flakiness.

4 RQ_2. THE COMBINED EFFECTS OF METRICS ON TEST FLAKINESS

4.1 Research Methodology

After studying the statistical significance of the distributions of our independent variables, we proceeded with our second research question. We devised a *Logistic Regression Model*, which belongs to the class of *Generalized Linear Model* (GLM) [25]. We have used this statistical modeling approach because it does not assume the distribution of data to be normal. In fact, we verified the normality of the distribution by means of the K-S Lilliefors test [10], which failed to reject the null-hypothesis, i.e., our data is not normally distributed. Furthermore, the *Logistic Regression Model* can deal with dichotomous dependent variables, hence fitting our case.

More formally, let $Logit(\pi_f)$ be the explained test flakiness f, let β_0 be the log odds of the likelihood of flakiness being increased in a test, and let the parameters $\beta_1 \cdot f_1$, $\beta_2 \cdot f_2$, ..., $\beta_n \cdot f_n$ be the differentials in the log odds of being the likelihood of flakiness increased for a test with characteristics f_1, f_2, ..., f_n, the statistical model is represented by the function:

$$Logit(\pi_f) = \beta_0 + \beta_1 \cdot f_1 + \beta_2 \cdot f_2 + \cdots + \beta_n \cdot f_n. \quad (1)$$

To implement the model, we relied on the `glm` function available in R toolkit.[2] Moreover, to avoid multi-collinearity we used the `vif` (Variance Inflation Factors) function implemented in R to discard non-relevant variables, putting a threshold value equal to 5 [26].

4.2 Analysis of the Results

Table 3 reports the results of the *Logistic Regression Model*. The table reports only 17 of the independent variables; the other eight factors, i.e., Halstead Length, Halstead Volume, LOC, MPC, RFC, WMC, Complex Class, and Spaghetti Code, have been excluded by the model as a result of the `vif` analysis. For each variable, the table reports the value of the estimate, the standard error, and the statistical significance. The latter is explained by the number of stars, i.e., '***' indicates a p<0.001, '**' indicates a p<0.01, '*' indicates a p<0.05 and '.' indicates a p<0.1.

The results confirm the discussion drawn in the context of $\mathbf{RQ_1}$, but not always the factors useful to distinguish flaky and non-flaky tests. We can confirm the role of code complexity: while some of the metrics whose distribution differs in flaky and non-flaky tests were excluded by the `vif` analysis, we can still see Halstead Vocabulary and other metrics connected to complexity, i.e., lines of test code and CBO, as statistically significant.

Similarly, the assertion density keeps being statistically significant. In addition, the logistic modeling let *Assertion Roulette* become significant: this test smell arises when a test has a number of undocumented assertions that would not allow a developer to properly understand the rationale behind a failure [36]. Of course, the connection with the flakiness might be indirect and due to a reflection of the assertion density: we plan to further investigate the role of this test smell in our future research on the matter.

Last but not least, our results indicate that two code smells such as *Functional Decomposition* and *Class Data Should be Private* are statistically significant. Looking at the definitions, these smells have not obvious connections to flakiness. The first arises when a class does not follow object-oriented principles like polymorphism and inheritance, declaring a few methods that look like procedural functions. The second affects classes that do not encapsulate fields, hence providing public access to their attributes. To provide an interpretation of this finding, we manually dived into the dataset and analyzed a sample of the production classes affected by those smells. In particular, we randomly selected 20 classes affected by each smell and tried to establish a motivation for the statistical results obtained—this process was mainly conducted by the first author of the paper, who was supported by the other authors whenever needed. As a result, we could discover that the examined classes had high cyclomatic complexity and, most likely, the two smells statistically subsumed the other complexity metrics. In other words, it

[2] https://www.r-project.org/

Table 3: Results for RQ_2 achieved by the statistical model.

Generalized Linear Model							
	Estimate	S.E.	Sig.		Estimate	S.E.	Sig.
Intercept	-4.82	2.93		Cond. Test Logic	-14.10	9.67	
TLOC	7.45	2.35	**	Fire and Forget	2.08	1.73	
McCabe	0.90	0.72		LCOM2	1.11	1.38	
Assertion Density	2.50	0.89	**	LCOM5	-0.83	1.39	
Assertion Roulette	-19.95	9.74	*	CBO	1.82	0.80	*
Mystery Guest	-0.60	2.71		Halstead Voc.	4.73	0.97	***
Eager Test	3.70	0.98	***	CDSBP	2.93	1.47	*
Sensitive Equality	-2.71	6.28		FD	0.75	0.28	**
Resource Optimism	-2.70	4.89		God Class	-1339.58	2621.51	

is not the presence of these code smells to directly influence the test flakiness, but rather a co-occurring phenomenon; code complexity was again confirmed to be the main distinguishing factor.

Key findings of RQ_2.

While some of the metrics that turned out to be significant in $\mathbf{RQ_1}$ were discarded, we could confirm that factors connected to code complexity and assertions represent the main distinguishing elements for test flakiness. In addition, the presence of some forms of code smells may be considered as a proxy measure to estimate the likelihood of tests to be flaky.

5 DISCUSSION AND LIMITATIONS

The results of our feasibility study provided insights that are worth to further discuss along with the limitations of our study.

5.1 Discussion

Our aim was to conduct a preliminary, feasibility study that could shed light on the potential features to be used within machine learning models that predict test flakiness by means of statically computable metrics and smells. Our findings revealed three main observations in this respect. In the first place, the role of code complexity: according to our statistical exercise, a number of metrics and smells that have to do with the complexity of production and test code turned out to significantly influence the test flakiness. This is a conclusion that we reached in both research questions, even though we observed different statistically relevant factors.

On the one hand, these results let emerge a potential additional property of complexity metrics, namely that of influencing the likelihood of tests to be flaky. We plan to investigate such an unknown relation further, yet it represents already a motivation for researchers and practitioners to keep studying and monitoring code complexity to improve source code quality and ease the test code development activities. On the other hand, our findings started addressing the problem of test flakiness in a more comprehensive manner: as a matter of fact, the properties of production code might affect the likelihood to observe an unreliable behavior in the tests.

The second key observation concerns with the role of assertions. To the best of our knowledge, only a few pieces of work [7] have investigated how `assert` statements can induce test flakiness. As such, further investigations into the matter might provide additional insights and instruments to deal with flaky tests.

Finally, the role of code and test smells. As for the former, different design issues were found significant in $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$. We could delineate direct and indirect connections that should be further analyzed: in any case, our results suggest that, in some cases, code smells might subsume properties that might support the identification of flaky tests. As for the latter, the discussion is slightly different and perhaps more interesting. Our study could not investigate the role of test smells in an organic manner, as certain types of smells were not detected in the exploited dataset. Nonetheless, we could still discover a relation between flaky tests and smells like *Assertion Roulette* and *Eager Test*, hence paving the way for confirmatory, larger-scale investigations.

All in all, we can conclude our feasibility study by highlighting the concrete possibility to devise fully static mechanisms for detecting flaky tests. This is, clearly, our followup step.

5.2 Threats to Validity

When it comes to the limitations of the study, there are some factors that might have biased our conclusions.

Construct Validity. Threats in this category refer to the correctness of the dataset used in the study. We relied on a publicly available source built in the context of previous research [16] and that has been already used and validated. This makes us confident of the reliability of the dataset; yet, we cannot exclude imprecision, especially in terms of the flaky tests identified, e.g., some tests might have not exposed their unreliability over the multiple executions performed by the authors of the dataset.

To compute the independent variables, we relied on automated tools. Also in this case, we cannot exclude imprecision, especially in terms of code and test smells output by the employed detectors. To partially mitigate this threat, we selected tools that have been previously evaluated, showing good accuracy.

Conclusion Validity. As for the statistical methods employed, we selected the Generalized Linear Model after verifying its suitability for our purpose, e.g., its ability to deal with dichotomous variables. In addition, to ensure that the model did not suffer from multi-collinearity, we applied a stepwise procedure, using the `vif` function, aimed at discarding non-relevant independent variables.

Another possible threat concerns with the unbalance of the dataset considered. As a matter of fact, the number of flaky tests was way lower than the one of non-flaky tests. This aspect might have particularly biased the metric distributions observed in $\mathbf{RQ_1}$. To account for this potential issue, we conducted an additional experiment. First, we randomly selected a number of non-flaky tests equal to the number of flaky tests: in this way, we could create a reduced version of the dataset that balanced the number of test cases in the two sets. Second, we re-run the entire set of experiments to verify whether the obtained results were in line with those discussed in the paper. This is what actually happened: for the sake of space limitations, the additional analyses are reported in our online appendix [33], yet we could confirm our findings, hence mitigating the threat to validity.

External Validity. With respect to the generalizability of the results, we are aware of the limitations of our study. We focused on the iDFlakies dataset [16], which is limited to open-source projects written in Java. Our future research agenda includes the extension of the study with other datasets of flaky tests.

6 RELATED WORK

Due to space limits, we cannot provide an extensive analysis of the literature on flaky tests. Therefore, we only discuss the seminal papers on the topic that have inspired our research.

Luo et al. [19] manually inspected 1,129 commits to elicit a taxonomy reporting ten root causes of test flakiness. Thorve et al. [35] conducted a similar study in Android apps, concluding that some root causes are similar to those identified by Luo et al. [19], while others relate to program logic and UI. Eck et al. [7] built upon these papers to identify additional root causes, shedding lights into the potential contribution provided by production code factors. When setting our study, we took the work by Eck et al. [7] into account and computed a number of production code metrics and smells.

In terms of prediction models, several researchers have investigated the role of the test code vocabulary, reporting promising results [3, 13, 32, 37]. In an alternative manner, Alshammari et al. [1] used a combination of static and dynamic metrics to predict flaky tests. With respect to these works, ours can be seen as complementary: we aimed at exploiting the metric profile of tests and their corresponding production code, rather than relying on combinations of features or source code terms.

7 CONCLUSION AND FUTURE WORK

We presented a feasibility study aimed at understanding whether statically computable metrics can be exploited for flaky test prediction. Our findings revealed that this seems to be possible: the complexity of production/test code, the distribution of `assert` statements, and the presence of code/test smells represent potentially valid indicators of test flakiness. Our future research agenda includes the replication of the study on more diverse datasets. We also plan to experiment with machine learners trained using the significant static metrics and smells from our analysis.

ACKNOWLEDGEMENT

Fabio gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090.

REFERENCES

[1] A. Alshammari, C. Morris, M. Hilton, and J. Bell. 2021. Flakeflagger: Predicting flakiness without rerunning tests. In *ICSE 2021*. IEEE, 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140
[2] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE 2018*. IEEE, 433–444. https://doi.org/10.1145/3180155.3180164
[3] B. Camara, M. Silva, A. Endo, and S. Vergilio. 2021. What is the Vocabulary of Flaky Tests? An Extended Replication. *arXiv preprint arXiv:2103.12670* (2021).
[4] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci. 2019. How the experience of development teams relates to assertion density of test classes. In *ICSME 2019*. IEEE, 223–234. https://doi.org/10.1109/ICSME.2019.00034
[5] S. Chidamber and C. Kemerer. 1994. A metrics suite for object oriented design. *IEEE TSE* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895
[6] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. 2009. ReAssert: Suggesting repairs for broken unit tests. In *ASE 2009*. IEEE, 433–444. https://doi.org/10.1109/ASE.2009.17
[7] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *ESEC/FSE 2019*. 830–840. https://doi.org/10.1145/3338906.3338945
[8] M. Fowler. 2011. Eradicating non-determinism in tests. *Martin Fowler Personal Blog* (2011).
[9] M. Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[10] G. Garson. 2012. Testing statistical assumptions. *Asheboro, NC: Statistical Associates Publishing* (2012).
[11] G. Grano, C. De Iaco, F. Palomba, and H. Gall. 2020. Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality. In *ICSME 2020*. IEEE, 336–347. https://doi.org/10.1109/ICSME46990.2020.00040
[12] G. Grano, F. Palomba, and H. Gall. 2019. Lightweight assessment of test-case effectiveness using source-code-quality indicators. *IEEE TSE* (2019). https://doi.org/10.1109/TSE.2019.2903057
[13] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon. 2021. A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests. In *MSR 2021*. https://doi.org/10.1109/MSR52588.2021.00034
[14] J. Han, M. Kamber, and J. Pei. 2011. Data mining concepts and techniques third edition. *The Morgan Kaufmann Series in Data Management Systems* 5, 4 (2011), 83–124.
[15] F. Lacoste. 2009. Killing the gatekeeper: Introducing a continuous integration system. In *2009 agile conference*. IEEE, 387–392. https://doi.org/10.1109/AGILE.2009.35
[16] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST 2019*. IEEE, 312–322. https://doi.org/10.1109/ICST.2019.00038
[17] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *ISSRE 2020*. IEEE, 403–413. https://doi.org/10.1109/ISSRE5003.2020.00045
[18] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428270
[19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An empirical analysis of flaky tests. In *ESEC/FSE 2014*. 643–653. https://doi.org/10.1145/2635868.2635920
[20] T. McCabe. 1976. A Complexity Measure. *IEEE TSE* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837
[21] A. Memon and M. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *ICSE 2013*. IEEE, 1479–1480. https://doi.org/10.1109/ICSE.2013.6606750
[22] J. Micco. 2017. The state of continuous integration testing@ Google. (2017).
[23] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE TSE* 36, 1 (2009), 20–36. https://doi.org/10.1109/TSE.2009.50
[24] J. Murillo-Morera and M. Jenkins. 2015. A Software Defect-Proneness Prediction Framework: A new approach using genetic algorithms to generate learning schemes. In *SEKE*. 445–450. https://doi.org/10.18293/SEKE2015-099
[25] J. Nelder and R. Wedderburn. 1972. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)* 135, 3 (1972), 370–384. https://doi.org/10.2307/2344614
[26] R. O'brien. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity* 41, 5 (2007), 673–690. https://doi.org/10.1007/s11135-006-9018-6
[27] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221. https://doi.org/10.1145/3180155.3182532
[28] F. Pecorelli, G. Di Lillo, F. Palomba, and A. De Lucia. 2020. VITRuM: A Plug-In for the Visualization of Test-Related Metrics. In *AVI 2020*. 1–3. https://doi.org/10.1145/3399715.3399954
[29] F. Pecorelli, F. Palomba, and A. De Lucia. 2021. The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems. *Empirical Software Engineering* 26, 2 (2021). https://doi.org/10.1007/s10664-020-09891-y
[30] A. Perez, R. Abreu, and A. van Deursen. 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *ICSE 2017*. IEEE, 654–664. https://doi.org/10.1109/ICSE.2017.66
[31] M. Pezze and M. Young. 2008. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons.
[32] G. Pinto, B. Miranda, S. Dissanayake, M. D'Amorim, C. Treude, and A. Bertolino. 2020. What is the vocabulary of flaky tests?. In *MSR 2020*. 492–502. https://doi.org/10.1145/3379597.3387482
[33] V. Pontillo, F. Palomba, and F. Ferrucci. 2021. Toward Static Test Flakiness Prediction: A Feasibility Study. https://doi.org/10.6084/m9.figshare.14645895.v3
[34] V. Terragni, P. Salza, and F. Ferrucci. 2020. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In *ICSE 2020*. 69–72.
[35] S. Thorve, C. Sreshtha, and N. Meng. 2018. An empirical study of flaky tests in android apps. In *ICSME 2018*. IEEE, 534–538. https://doi.org/10.1109/ICSME.2018.00062
[36] A. van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. 2001. Refactoring test code. In *XP 2001*. Citeseer, 92–95.
[37] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino. 2021. Know You Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021), 76119–76134. https://doi.org/10.1109/ACCESS.2021.3082424
[38] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA 2014*. 385–396. https://doi.org/10.1145/2610384.2610404