The Vocabulary of Flaky Tests in the Context of SAP HANA

Alexander Berndt
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
0009-0009-5248-6405

Zoltán Nochta
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
0000-0001-9146-8968

Thomas Bach
SAP
Walldorf, Germany
0000-0002-9993-2814

***Abstract*—Background. Automated test execution is an important activity to gather information about the quality of a software project. So-called flaky tests, however, negatively affect this process. Such tests fail seemingly at random without changes to the code and thus do not provide a clear signal. Previous work proposed to identify flaky tests based on the source code identifiers in the test code. So far, these approaches have not been evaluated in a large-scale industrial setting.**

Aims. We evaluate approaches to identify flaky tests and their root causes based on source code identifiers in the test code in a large-scale industrial project.

Method. First, we replicate previous work by Pinto et al. in the context of SAP HANA. Second, we assess different feature extraction techniques, namely TF-IDF and TF-IDFC-RF. Third, we evaluate CodeBERT and XGBoost as classification models. For a sound comparison, we utilize both the data set from previous work and two data sets from SAP HANA.

Results. Our replication shows similar results on the original data set and on one of the SAP HANA data sets. While the original approach yielded an F1-Score of 0.94 on the original data set and 0.92 on the SAP HANA data set , our extensions achieve F1-Scores of 0.96 and 0.99, respectively. The reliance on external data sources is a common root cause for test flakiness in the context of SAP HANA.

Conclusions. The vocabulary of a large industrial project seems to be slightly different with respect to the exact terms, but the categories for the terms, such as remote dependencies, are similar to previous empirical findings. However, even with rather large F1-Scores, both finding source code identifiers for flakiness and a black box prediction have limited use in practice as the results are not actionable for developers.

***Index Terms*—test flakiness, software testing, regression testing, machine learning**

I. Lay Abstract

We train multiple machine learning models to predict whether a test is flaky or not. As a training input, we use source code identifiers in the test code and historical test results. Furthermore, we identify the features with the highest information gain for test flakiness. Broadly speaking, the keywords that have the highest impact to trigger flakiness.

The models on data from SAP HANA achieve F1-Scores of 75% and 94%. The first score for data collected from CI, the second from data collected from a dedicated flakiness experiment. Our results also show trade-offs between techniques with higher investments in training time versus F1-Score gains.

[0] Source code available via: https://doi.org/10.5281/zenodo.8107295

978-1-6654-5223-6/23/$31.00 ©2023 IEEE

Our results show that *virtual* is the identifier most strongly associated with test flakiness. In context of SAP HANA, *virtual* refers to virtual tables, which represent tables from remote sources. That shows an important practical limitation. We cannot advise developers to stop using *virtual*. Although we know *virtual* has a high chance of leading to flakiness, developers also have to test exactly this functionality. Similarly, if we would use the trained model as a prediction for code changes, the results would not be actionable for developers. They would just get a probability value that their change can lead to flakiness, but no clear instructions what to improve. Therefore, while we can replicate previous results, we conclude that the practical usefulness is limited so far.

II. Introduction

With the rise of agile software development and shorter release cycles, companies aim for continuous integration (CI) and continuous delivery (CD) [1]. This often requires regression tests to verify that a change does not cause unintended effects [2]. A change can only be automatically integrated into the main code line if all tests show positive results. However, this process is commonly affected by so-called flaky tests. A flaky test occasionally fails seemingly at random, without changes to the executed code. With that, a test may not provide a clear signal and indicate an issue, which does not exist in the code. Thus, flaky tests interfere with the automatic integration of code changes and complicate quality assurance.

Flaky tests are a common problem in the software industry [3]–[6]. In the context of large-scale software, where automation has an increasingly important role, test flakiness negatively affects automation and quality assurance. Thus, software companies put a lot of effort into handling flaky tests [4], [7]–[10].

Previous research shows that a common strategy for dealing with flaky tests is re-executing failed test executions and accepting the test result if one re-execution passes [11]. The CI system for SAP HANA utilizes, among others, the same strategy. When a test case fails, it is re-executed three times. If one of the additional test executions reports a passing result, the test is considered successful [9], [12]. However, re-executing tests is costly with regard to computational resources. Furthermore, even though the additional test executions can be executed in parallel, the step *re-executing*

tests happens after the first execution and is therefore a sequential activity. Such a sequential step negatively affects the overall CI/CD process and increases waiting times for developers considerably. Hence, reducing the number of flaky failures could save computational resources and decrease the turnaround time of integrating code changes.

Debugging flaky tests is a complex task for developers, because flaky failures might not be reproducible by re-executing the test [13]. Therefore, previous work has investigated detecting flaky tests statically, i.e., without relying on (re-)executions. Identifying source code and with that the root causes of test flakiness might help developers to fix and avoid flaky tests in the future [9].

Recent approaches to detect flaky tests have focused on the use of machine learning algorithms. Assuming that test code yields information about the root cause for the flaky behavior of a test, Pinto et al. propose to detect flaky tests based on the source code identifiers in the test code with promising results [14]. Additionally, by analyzing the employed features for the prediction, they identify source code identifiers associated with flaky tests. These source code identifiers provide information on the underlying root cause for the flakiness. To the best of our knowledge, this approach has not been evaluated in a large-scale industrial context yet. We provide such an evaluation in the context of SAP HANA via replication of previous work [15]. Overall, our contributions are:

1) A replication of previous work to identify the vocabulary of flaky tests for a large industrial software project.
2) The evaluation of multiple techniques for predictingthe flakiness of tests. Our comparison includes TF-IDF, TF-IDFC-RF, CodeBERT, and XGBoost.
3) Discussion of the practical usefulness of the previously mentioned approaches and their findings.

III. Research Questions and Previous Work

In this section, we introduce the research questions and relate them to the findings of previous studies.

RQ1: By replicating previous work of Pinto et al. [14], what F1-Scores do we achieve for the original data set and two data sets collected for SAP HANA?

Under the assumption that the source code of flaky tests follows certain syntactical patterns, Pinto et al. tried to predict flakiness automatically with the help of Natural Language Processing (NLP) techniques [14]. They evaluated their model on one of the largest data sets for flaky tests obtained by DeFlaker [16]. The DeFlaker data set consists of over 5000 flaky tests [17]. However, as the DeFlaker data set does not contain any non-flaky tests, Pinto et al. re-executed the tests from the different projects in the DeFlaker data set 100 times and labeled tests as non-flaky when they showed consistent results over the 100 executions. In the end, a data set containing approximately 1400 flaky and non-flaky tests was obtained. To analyze the test code, Pinto et al. extracted source code identifiers and split them using their camel-case syntax. After that, they performed common pre-processing steps like stemming and stop word removal and embedded the code identifiers in a bag-of-words. Furthermore, they added the number of Java keywords and the number of lines of code to the feature set to incorporate a proxy for code complexity. Subsequently, the pre-processed data was employed to train several machine learning models, receiving the best F1-Score of 0.95 with a random forest classifier [18]. We provide an evaluation of Pinto et al.'s study in the context of SAP HANA.

RQ2: What are the most important features of the prediction model and how do they reflect the most prevalent root causes of test flakiness at SAP HANA?

Pinto et al. also identified the test code identifiers that are most strongly associated with flakiness. To achieve this, they calculate the information gain from the incorporated features. With that, they could identify the vocabulary of flaky tests in the DeFlaker data set. The resulting vocabulary mainly consisted of words related to remote task execution and event queues. Aside from that, a high number of occurrences of the Java keyword *throw* decreased the likelihood of a flaky test. Pinto et al. conclude that enforcing proper exception handling in the test code can be a measure to prevent test flakiness. In our work, we calculate the information gain of the test code identifiers in two data sets from SAP HANA to identify the most prevalent root causes of test flakiness.

RQ3: How do the extensions TF-IDF, TF-IDFC-RF, XGBoost, and CodeBERT compare against the original approach of Pinto et al. in terms of F1-Score?

TF-IDF and TF-IDFC-RF for feature embedding: *Term Frequency - Inverse Document Frequency* (TF-IDF) is a term weighting scheme to represent text as a vector [19]. TF-IDF aims at improving Bag-of-Words representations with regard to classification tasks. Therefore, we compare the use of TF-IDF matrices in terms of prediction performance against the original approach using Bag-of-Words. To further advance the weightings of conventional TF-IDF for supervised classification tasks, previous research has proposed a wide range of approaches to perform so-called supervised term weighting [20]–[25]. We evaluate the supervised weighting scheme Term Frequency-Inverse Document Frequency in Classes-Relevance Factor (TF-IDFC-RF) as proposed by Carvalho et al. as this scheme showed the highest results on two benchmarks [26].

XGBoost as a classification model: Previous research recommends XGBoost in a wide variety of problems [27]. Thus, we evaluate XGBoost as an alternative implementation.

CodeBERT as a classification model: Fatima et al. used CodeBERT to predict test flakiness based on the source code of the test [28], [29]. Their evaluation suggests comparable good results on the *FlakeFlagger* data set [30]. Thus, we compare CodeBERT against the approach by Pinto et al. [14].

IV. Background and Data Sets

In this section, we introduce SAP HANA, the main subject of our study. SAP HANA is a large-scale in-memory database management system with millions of lines of code, mainly written in C++ and developed by SAP [31]–[33].

A. Testing at SAP HANA

The code of SAP HANA is tested by about one million tests, mostly written in C++ or Python [12]. Roughly speaking, each test can be classified as a unit test or system test. Hereby, unit tests are typically written in C++, whereas system tests are written in Python. In addition to Python code, a large portion of the system tests contain SQL statements to communicate with the SAP HANA instance under test [12]. To train the models in this work, we focus on system tests, as they are more expensive with regard to required hardware, software, and human effort [34].

For each test case execution of SAP HANA's test suite, SAP collects metadata and stores it in an internal SAP HANA instance. Among other attributes, this metadata includes the results of each test execution, allowing us to derive flaky labels based on the result history of tests.

B. Testing Stages

To assure the quality of SAP HANA, multiple testing stages varying in scope, effort, and frequency have been defined. Figure 1 provides an overview of the different stages. SAP collects most of the metadata about test results in three of these stages: pre-submit testing, post-submit testing, and extended testing. For example, within July 2022 alone, more than 800 million test case execution results have been reported. This amount of data enabled us to collect two flaky test data sets from SAP HANA: *Mass Test Execution 2020* (MTE20) and *Flaky Test 2021* (FT21).

C. MTE20

The so-called *"Mass Test Execution"* was conducted in 2020 and refers to a research project in which a subset of SAP HANA's tests was executed 100 times against the same build of the code. Hereby, all test results were persisted and thus were available for our work. Based on these test results, 35 000 test cases could be labeled as flaky or not.

For the labeling, we calculated the average result $\overline{x}_s$ of each test case s over the 100 executions. Hereby, 0 referred to a passing and 1 to a failing execution. A test case was labeled as flaky, when $\overline{x}_s \notin \{0, 1\}$.

D. FT21

The purpose of the FT21 data set was to examine whether our adopted approach is also applicable to data from SAP HANA's daily business. Therefore, we collected the data from test results from the CI system of SAP HANA in 2021. As mentioned before, failing tests in CI runs are re-executed three times for the same code. Similar to our labeling for MTE20, we labeled a test as flaky, when it yielded different results throughout these four executions.

However, since SAP HANA's tests are executed in a permanently evolving environment, test failures can be caused by so-called "global issues", i.e., problems that were not caused by the test code. For example, in July 2021, global issues affected roughly 10% of the builds in SAP HANA's CI pipeline. In many cases, such global issues caused flaky labels even though the failure was not related to the actual test code. However, in the context of our work, these labels could be considered noise and interfere with the training of our model [35]. Hence, we applied a heuristic approach to filter failures which were caused by global issues.

Based on the assumption that global issues affect multiple tests simultaneously, the following filter was applied: Let s_i be the number of passing test cases and n_i the total number of executed tests for build i of SAP HANA.

To filter out builds that were affected by a global issue, we only consider tests from builds that fulfill both of the following conditions:

(i) $\frac{s_i}{n_i} > 0.99$
(ii) $n_i > 1000$

That is, we took only test results from builds with at least 1,000 executed tests into account, of which at least 99% have been successful. By doing so, we obtained labels for approximately 15,000 test cases.

E. MSR4Flakiness

To compare the results of our replication setup and its extensions against the results from the original study, we employed the MSR4Flakiness data set in addition to the two data sets from SAP HANA. Pinto et al. collected the data for the MSR4Flakiness data set for their work [14], [36].

F. Exploratory Data Analysis

To amplify our intuition about the available data from SAP HANA, we first conducted a manual examination of the test code with regard to test flakiness [37]. Therefore, we assessed commits, which were supposed to fix flaky labeled bugs. To compare the results against findings from previous literature, commits were divided into the categories proposed by Luo et al. [38] as shown in Table I. Besides, we added the category *Fixed Timeout*, because increasing the maximum duration of a test was one of the most prevalent fixes for flaky behavior in the context of SAP HANA. Hereby, the maximum duration refers to the maximum execution time of a test before it is cancelled. Table I shows the results of the analysis.

The three most prevalent categories for test flakiness in the assessed commits are *Fixed Timeout*, *Concurrency* and *Async Wait*. These findings are in line with the results provided by previous work. Luo et al. report the same categories [38]. A noticeable difference to previous findings are large number of commits in the category *Hard to classify*. However, this gap can be explained by differences in the methodology. Luo et al. time-boxed their approach to 2 hours per commit, whereas we used an overall time-box of 8h, which equals to roughly 10 minutes per commit. That means we spent less time per commit for classification and our project might be larger and therefore more complex to find a definitive classification.

V. Replication Setup

As described in Section II, we conduct a conceptual replication of a previous study [14], [39]. This section provides an overview of the steps conducted to implement our replication.

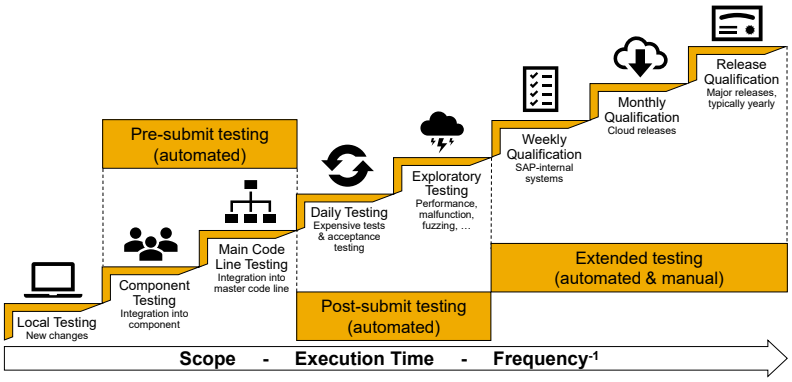

Fig. 1: Testing Stages of SAP HANA

TABLE I: Root Causes of Flaky Tests

Root Cause	Amount	Percentage
Fixed Timeout	8	17%
Concurrency	7	15%
Async Wait	6	11%
Test Order Dependency	6	11%
Unordered Collections	3	6%
Randomness	2	4%
Resource Leak	1	2%
Time	1	2%
Hard to Classify	13	28%
Network	0	0%
IO	0	0%
Total	47	

TABLE II: Distribution of Classes Across Data Sets

Data Set	Total	Non-Flaky	Flaky
FT21	14,703	13,566	1,137
MTE20	36,963	34,311	652
MSR4Flakiness	45,381	44,011	1,370

A. Data Resampling

A common problem with data related to test flakiness is the imbalance between non-flaky and flaky samples [30]. Typically, data sets contain a higher number of non-flaky samples [40]. This problem applies to our data sets as well. Table II displays the class distribution of the data sets, showing that all of them contain more samples from the non-flaky class. This leads to the conclusion that all of them are imbalanced.

Using an imbalanced data set to train a machine learning classifier can be problematic as the classifier might be skewed towards the majority class. That is, in the context of the data sets for this work, the classifier could overestimate the probability that a sample is non-flaky.

A common technique to handle imbalanced data is *undersampling* [41]. Hereby, the data set is balanced out by keeping all the samples from the minority class, whilst decreasing the size of the majority class by randomly dropping samples. For this work, we implemented undersampling with *RandomUnderSampler* [42] provided by *imbalanced-learn* [42]. We employed undersampling as the first step of our pipeline, as it reduces the total amount of samples to be processed, thus saving computational resources for future steps.

B. Tokenization

Manning et al. define tokenization as the task to chop a document into pieces [43]. They call the resulting pieces tokens and state that "tokens are often loosely referred to as terms or words" [43]. Tokenization can also involve the removal of unwanted characters, for example, punctuation. For this work, we processed tokens depending on their type and content as defined by the *token* library [44]:

1) **Keywords**: As Python keywords should be treated differently than source code identifiers in a later step of the pipeline, we identified and marked them during tokenization. We also add the token *self* to the list of keywords. Although *self* is not a keyword in Python, it is conventionally used.
2) **Numbers**: Previous research has shown that one of the most prevalent categories for test flakiness is *Async Wait* [38]. For example, in the context of SAP HANA, we found flaky tests, which use *time.sleep(n)* to wait *n* milliseconds for an external task to finish. To grasp all these occurrences in one token, numbers were masked with the *#NUM#* token.
3) **Names**: Tokens of type *NAME* were added to the resulting list of tokens.
4) **String**: Tokens of type *STRING* were tokenized under the use of *word_tokenize* as provided by *nltk* [45].

Figure 2 shows the resulting tokens for an exemplified test after we apply the described steps. In contrast to the study by Pinto et al., we did not apply stemming and stop word removal, as the results of Pinto et al. showed that these steps did not have any effect on the resulting model.

C. Identifier Splitting

The study by Pinto et al. has shown that identifier splitting can have a positive impact on the performance of a vocabulary-

```
def testBow(self):
    """ exemplified code """
    self.remoteServer.connect()
    result = self.waitForRandomStringFromRemote(10)
    self.assertTrue(not result.isnumeric())
    self.assertTrue(len(result) > 0)
```

```
#KEYWORD#_def testBow test Bow #KEYWORD#_self #KEYWORD#_self
remoteServer remote Server connect result #KEYWORD#_self
waitForRandomStringFromRemote wait For Random String From
Remote #NUM# #KEYWORD#_self assertTrue assert True
#KEYWORD#_not result isnumeric is numeric #KEYWORD#_self
assertTrue assert True len result #NUM#
```

Fig. 2: Example of Tokenization

TABLE III: Results of Ronin Algorithm

Input String	Resulting Tokens
testBow	['test', 'Bow']
remoteServer	['remote', 'Server']
assertTrue	['assert', 'True']

based classifier for test flakiness prediction [14]. Identifier splitting refers to the task of tokenizing source code identifiers. A common way to achieve this is to split source code identifiers based on their case style [46]. For example, the identifier *waitForSomething*, which is written in *"camelCase"*, can be chopped up into the tokens *wait*, *For*, and *Something*.

In the original study, Pinto et al. split the identifiers based on their camel-case syntax. However, in the context of SAP HANA, we found that the test code contains different casing styles. To split source code identifiers based on different casing styles, Hucka et al. introduced the package *Spiral* [47]. The package Spiral implements a range of splitting algorithms such as the Ronin algorithm. The Ronin algorithm was already employed in the context of SAP HANA in a previous study [48]. Thus, we considered it suitable as a tokenizer to split identifiers for this work. Table III shows the resulting split identifiers from Figure 2 under the use of Spiral and Ronin. As proposed by Pinto et al. [14], all the split tokens together with the original identifier were added to the result set.

D. Feature Extraction

To enable a machine learning classifier to learn from data, the characteristics of the data have to be captured in a set of vectors [49]. We evaluate three approaches to achieve this: Bag-of-Words, TF-IDF, and TF-IDFC-RF.

As proposed by Camara et al. [50], we employ the *CountVectorizer* class from *Scikit-Learn* to implement Bag-of-Words. For TF-IDF, the respective *TfIdfVectorizer* class was used. The implementation of TF-IDFC-RF in this work is based on the Python implementation provided by Carvalho et al. [26], [51]. In this implementation, the *Tfidfcvectorizer* class is provided, which implements fit and transform methodology similar to classes from *Scikit-Learn*.

TABLE IV: Results Replication SAP HANA

Data Set	Precision	Recall	F1-Score
MSR4Flakiness	0.94	0.94	0.94
FT21	0.76	0.76	0.75
MTE20	0.97	0.87	0.92

E. Evaluation

To train and evaluate a machine learning classifier, it is common practice to split the available data into separate training and test data [14], [28], [30]. To achieve valid evaluation results, it is important to avoid data leakage, i.e. utilizing the test data during the learning setup. For example, Arp et al. suggest that a common mistake in the context of NLP tasks is to compute TF-IDF weightings on the complete data set instead of deriving them only from the training data [52]. To avoid such mistakes, we split the data for this work before embedding the code into feature vectors.

To evaluate the classifiers for this work, we conduct k-fold validation with $k = 5$ folds under the use of the *KFold* class provided by *Scikit-Learn* [53].

F. CodeBERT

In contrast to the other evaluated extensions, we set up a different preprocessing and evaluation pipeline for CodeBERT. To predict whether a test is flaky or not, CodeBERT does not require prior feature extraction but captures potential patterns in the source code automatically. With regard to evaluation, we did not use k-fold cross-validation, because of the increased computational cost to train CodeBERT. Instead, we employed a train-test-split, using 80% of the data for training and 20% as a test set.

VI. Results

In this section, we answer the research questions.

A. Research Question 1: Flaky Test Detection Benchmark

We replicate the setup of vocabulary-based models as proposed by previous studies and evaluate the replication on the two data sets from SAP HANA [14], [50], [54]. We verify the implemented pipeline and models for this work by comparing our results against previous benchmarks on the MSR4flakiness data set [36]. For the evaluation, we use 5-fold cross-validation, i.e., we trained and evaluated every model five times. We obtain the final result by rounding the results of the five runs to the second decimal and calculating the arithmetic mean. Tables IV and V show the respective results.

The replication yields similar results on the MSR4flakiness benchmark compared to previous studies [14], [50]. The results for the two data sets from SAP HANA differ. While the result on the MTE20 data set is similar to the result on the MSR4Flakiness data set, the result on the FT21 data set

TABLE V: Comparison to Original Approach

Approach	Precision	Recall	F1-Score
Original	0.99	0.91	0.95
Replication	0.94	0.94	0.94

TABLE VI: Top Features by Information Gain for FT21

Feature	Information Gain
virtual	0.039
esh	0.029
expectedresult	0.025
_runtest	0.023
doquery2	0.022
doexplain	0.020
virtual_product	0.020
uvalue	0.019
doexecute	0.019
sameresults	0.019

is worse. However, this difference may be explained by the findings of Haben et al. [54]. In their replication of Pinto et al.'s study, they found that vocabulary-based models suffer from a time-sensitive evaluation. That is, the performance of their models dropped when they used code from older revisions for training a model to predict flakiness for code from newer revisions. Likewise, the FT21 data set contains commits from a time frame of one year, i.e., the data is based on a range of revisions. In contrast, the data for the MTE20 data set contains only one revision.

Another possible explanation is that we collected the FT21 data set from test results of SAP HANA's CI pipeline, while the data in the MTE data set was purposefully generated to gain insights about test flakiness. Thus, the FT21 data set could be more strongly affected by noise than the MTE20 data set.

Conclusion: Our replication shows an F1-Score of 94% for MTE20 and 75% for FT21.

TABLE VII: Top Features by Information Gain for MTE20

Feature	Information Gain
virtual	0.239
doquery2	0.187
explain	0.169
do	0.165
doexecute	0.158
sameresults	0.156
sameexplain	0.121
same	0.120
dxchg	0.113
doexplaindxchg	0.113

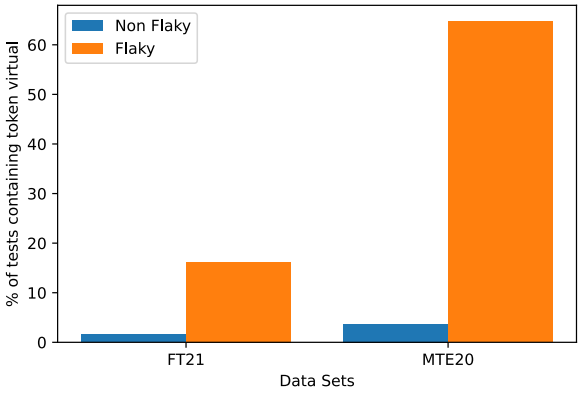

Fig. 3: Occurrences of *virtual*

B. Research Question 2: Detect Root Causes

The second research question aims at finding the root causes of test flakiness in context of SAP HANA. To achieve this, we calculate the information gain from the different features as proposed by Camara et al. [50]. Tables VI and VII show the 10 most important features for the FT21 and MTE20 data set, respectively. Comparing Table VI and Table VII, the best features for both data sets are similar . Although in a different order, 4 out of the 10 best features are present for both data sets . For both data sets, the most important feature is *virtual*. Looking at the underlying code, the most common usage of the token *virtual* in SAP HANA's tests is in the context of so-called virtual tables. Virtual tables provide access to tables on a remote source [55]. This finding is in line with previous research, which points out that the reliance on remote sources is a typical root cause for test flakiness [9], [14], [38].

To underline the extent of this finding, we count the number of flaky samples containing *virtual*. As shown in Figure 3, *virtual* appears in approximately 65% of the flaky samples, but only in 4% of the non-flaky samples from the MTE20 data set. In absolute numbers, 422 out of 652 flaky samples in the MTE20 data set contain the token *virtual*. However, the token is only contained in 24 non-flaky samples. Predicting a test to be flaky if *virtual* is present would result in a precision of approximately 95%. In the FT21 data set, *virtual* appears in approximately 16% (184 out of 1.137) of the flaky samples and 1.6% (18 out of 1.137) of the non-flaky samples. A prediction based solely on the appearance of *virtual* would result in a precision of approximately 91%.

Since the vocabulary of flaky samples appears similar in both data sets, we can conclude that both data sets contain a similar set of flaky samples. Although the labeling strategy and the examined time frame of the two data sets are completely different, they share 121 flaky labeled test cases. The vast majority of these tests verify the data federation functionality of SAP HANA, which is closely related to virtual tables.

In addition to *virtual*, the two data sets share three additional top predictors, namely *doquery2*, *sameresults*, and *sameex-*

Model	BoW	TF-IDF	TF-IDFC-RF
Random Forest	**0.94**	**0.94**	**0.94**
XGBoost	0.92	**0.94**	**0.94**
Randomized	0.48	0.48	0.48
Only True	0.34	0.34	0.34
Only False	0.32	0.32	0.32

TABLE VIII: Resulting F1-Scores for MSR4Flakiness

Model	BoW	TF-IDF	TF-IDFC-RF
Random Forest	0.75	**0.78**	**0.78**
XGBoost	0.71	**0.75**	**0.75**
Randomized	0.48	0.48	0.48
Only True	0.34	0.34	0.34
Only False	0.32	0.32	0.32

TABLE IX: Resulting F1-Scores for FT21

plain. Looking at the occurrences of these tokens, it appears that the three tokens are connected. The token *doquery2* refers to a function which implements the functionality to execute a query on the SAP HANA instance under test. The tokens *sameresults* and *sameexplain* refer to parameters of *doquery2*. As the *doquery2* function is defined in a superclass of the test cases for data federation, these tokens are also interconnected to the token *virtual*. In fact, all top predictors of the MTE20 data set arise from the context of data federation.

Regarding the FT21 data set, we did not examine every top predictor due to their low level of information gain.

Conclusion: The token *virtual* is the feature, which is most associated with flakiness in both data sets from SAP. Similarly, remote dependencies are a common root cause for flaky tests in both data sets from SAP.

C. Research Question 3: Evaluate Extensions

For the third research question, we evaluate TF-IDF and TF-IDFC as alternative term weighting schemes. In addition, we evaluate XGBoost as an alternative classification model.Tables VIII and X show the results for the evaluation runs. The values in the table depict the averaged F1-Scores from the 5-fold cross-validation. The results in Tables VIII and X show that the use of term weighting schemes instead of Bag-of-Words yields similar or better performance for every classifier on every data set. However, using supervised term weightings (TF-IDFC-RF) does not result in a better performance when compared to conventional TF-IDF.

Regarding model selection, XGBoost did not yield improved results when compared to Random Forest. In contrast, Random Forest outperformed XGBoost on all three data sets.

Finally, we fine-tune and evaluate CodeBERT on the three available data sets. As shown in Table XI, CodeBERT outperforms the previous approaches on every data set, providing state-of-the-art results on the MSR4flakiness data set.

However, while CodeBERT increases the previous prediction performance, it comes with the drawback of higher computational cost and runtime. Training and evaluating CodeBERT took over 10 times longer compared to random forest.

Conclusion: Using term weighting schemes improved the F1-Score with the largest improvement on the FT21 data set from 75% with Bag-of-Words to 78% with the two term weighting schemes TF-IDF and TF-IDFC-RF. CodeBERT has higher F1-Scores on each examined data set. The MTE20 data set shows the largest increase from 93% F1-Score to 99% F1-Score. However, the improvement in prediction performance requires at least a factor 10 increase in training time (10 min versus 2 hours). Furthermore, the resulting model requires over 100 times more disk space (4 versus 400 megabyte).

Model	BoW	TF-IDF	TF-IDFC-RF
Random Forest	0.92	**0.93**	**0.93**
XGBoost	0.91	**0.92**	**0.92**
Randomized	0.48	0.48	0.48
Only True	0.34	0.34	0.34
Only False	0.32	0.32	0.32

TABLE X: Resulting F1-Scores for MTE20

Data Set	CodeBERT	Previous Best
MSR4Flakiness	**0.96**	0.94
FT21	**0.82**	0.78
MTE20	**0.99**	0.93

TABLE XI: Resulting F1-Scores of CodeBERT

VII. Threats to Validity

A. Internal Validity

A possible threat to the internal validity of the conducted study lies in our data sets. Since we derived the labels from empirical observations, they could be affected by noise and might introduce bias to the results of this work. Since every test has a certain probability to fail occasionally, labeling tests as non-flaky, because they did not fail in a given time frame, could be considered a threat to validity. We mitigate this threat by employing two independent data sets from the same project.

Another possible threat to the internal validity are potential errors in the setup of the replication study. To minimize this threat, we compared our implementation against results from previous research [14]. The comparison shows similar results.

B. External Validity

The main threat to the external validity of this work is the unique context of SAP HANA. Although previous research evaluates a similar approach on other data sets, such approaches were not tested on a large-scale software project like SAP HANA. While the results of this work are in line with previous findings, we still consider it valuable to apply this study to further industrial and large projects. Thereby, we encourage studies on other larger data sets to further evaluate the generalizability of the presented approach.

C. Construct Validity

The main threat to the construct validity is our implementation to retrieve the test code of SAP HANA. We identify the test code for test results via an automated approach. Due to the complexity of the test suite and test configurations, this automated mapping might not be accurate. We manually verified several cases together with engineers from SAP.

VIII. Conclusion

We conceptually replicated previous work to detect flaky tests based on source code identifiers in the test code in a large-scale industrial project, namely SAP HANA. We compared the results of our replication against previous work on the original data set. With an F1-Score of 94%, our replication yields similar results compared to the original study (95%).

Our replication shows an F1-Score of 92% on the MTE20 data set of SAP HANA. We conclude that vocabulary-based models can detect flaky tests in a large-scale industrial setting.

Furthermore, we apply Pinto et al.'s approach to detect the root causes of flaky tests to SAP HANA's data sets. The presence of the token *virtual* in a test was the best predictor to distinguish between flaky and non-flaky tests. This leads to the conclusion that a common root cause for test flakiness at SAP HANA are virtual tables, which allow to access data from remote sources. This finding is in line with previous research, which considers the reliance on external resources the most common root cause for test flakiness [38], [56]–[58].

Finally, we evaluate various extensions to the original approach with regard to prediction performance. Hereby, we achieve the best result using CodeBERT with an F1-Score of 99% on the MTE20 data set. However, the use of CodeBERT comes with the drawback of increased computational cost.

From a practical point of view, the insights from this work are two-fold. On the one hand, we found that the trained models achieved better performance on MTE20. Therefore, we conclude that it is valuable to re-execute tests with the purpose of test flakiness examination. On the other hand, even though the trained models in this study yielded decent performance, we did not employ them in our developing process for two reasons. First, we expected the accuracy of the models to decay over time due to the ever-evolving code of SAP HANA. Second, testing the functionality of virtual tables is essential.

References

[1] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan, "On the Usage, Co-Usage and Migration of CI/CD Tools: A Qualitative Analysis," *Empirical Software Engineering*, 2023. DOI: 10.1007/s10664-022-10285-5.

[2] P. Bourque and R. E. Fairley, *SWEBOK: Guide to the Software Engineering Body of Knowledge*. 2014. [Online]. Available: http://www.swebok.org/.

[3] J. Listfield. "Where do our Flaky Tests Come From?" 2022-11-14, archived by Internet Archive at http://web.archive.org/web/20221113232600/https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html. (2020), [Online]. Available: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

[4] J. Palmer. "Test Flakiness - Methods for Identifying and Dealing with Flaky Tests." 2022-08-31, archived by Internet Archive at https://web.archive.org/web/20220831074314/https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/. (2019), [Online]. Available: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

[5] "Handling Flaky Tests at Scale: Auto Detection & Suppression." 2022-09-12, archived by Internet Archive at https://web.archive.org/web/20220912121107/https://slack.engineering/handling-flaky-tests-at-scale-auto-detection-suppression/. (2022), [Online]. Available: https://slack.engineering/handling-flaky-tests-at-scale-auto-detection-suppression/.

[6] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019. DOI: https://doi.org/10.1145/3293882.3330570.

[7] "Flaky Tests at Google and How We Mitigate Them." 2022-11-14, archived by Internet Archive at https://web.archive.org/web/20221113233026/https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. (2020), [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[8] "How do you Test your Tests?" 2022-11-14, archived by Internet Archive at https://web.archive.org/web/20221114080909/https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/. (2020), [Online]. Available: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/.

[9] O. Parry, G. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022. DOI: https://doi.org/10.1145/3476105.

[10] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and Ranking Flaky Tests at Apple," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020. DOI: https://doi.org/10.1145/3377813.3381370.

[11] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test Flakiness Across Programming Languages," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2039–2052, 2023. DOI: 10.1109/TSE.2022.3208864.

[12] T. Bach, A. Andrzejak, C. Seo, *et al.*, "Testing Very Large Database Management Systems: The Case of SAP HANA," *Datenbank-Spektrum*, Nov. 2022. DOI: 10.1007/s13222-022-00426-x.

[13] W. Lam, S. Winter, A. Astorga, and V. S. 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)*, 2020. DOI: https://doi.org/10.1109/ISSRE5003.2020.00045.

[14] 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. DOI: https://doi.org/10.1145/3379597.3387482.

[15] J. Carver, N. Juristo, M. Baldassarre, and S. Vegas, "Replications of software engineering experiments," *Empirical Software Engineering*, vol. 19, Apr. 2014. DOI: 10.1007/s10664-013-9290-8.

[16] 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*, 2018. DOI: https://doi.org/10.1145/3180155.3180164.

[17] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. "DeFlaker Website." 2022-08-15, archived by Internet Archive at https://web.archive.org/web/20220815084840/http://www.deflaker.org/icsecomp/. (2018), [Online]. Available: http://www.deflaker.org/icsecomp/.

[18] L. Breiman, "Random Forests," *Machine learning*, 2001. DOI: 10.1023/A:1010933404324.

[19] "TF-IDF," in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. 2010. DOI: https://doi.org/10.1007/978-0-387-30164-8_832.

[20] K. Chen, Z. Zhang, J. Long, and H. Zhang, "Turning from TF-IDF to TF-IGM for Term Weighting in Text Classification," *Expert Systems with Applications*, 2016. DOI: https://doi.org/10.1016/j.eswa.2016.09.009.

[21] F. Debole and F. Sebastiani, "Supervised Term Weighting for Automated Text Categorization," in *Proceedings of the 2003 ACM symposium on Applied computing*, 2004.

[22] M. Lan, C. L. Tan, J. Su, and Y. Lu, "Supervised and Traditional Term Weighting Methods for Automatic Text Categorization," *arXiv preprint arXiv:2003.07193*, 2009. DOI: https://doi.org/10.1109/TPAMI.2008.110.

[23] F. Ren and M. G. Sohrab, "Class-indexing-based Term Weighting for Automatic Text Classification," *Information Sciences*, 2013. DOI: https://doi.org/10.1016/j.ins.2013.02.029.

[24] T. Dogan and A. K. Uysal, "Improved Inverse Gravity Moment Term Weighting for Text Classification," *Expert Systems with Applications*, 2019. DOI: https://doi.org/10.1016/j.eswa.2019.04.015.

[25] J. Martineau and T. Finin, "Delta TF-IDF: An Improved Feature Space for Sentiment Analysis," in *Proceedings of the International AAAI Conference on Web and Social Media*, 2009.

[26] F. Carvalho and G. P. Guedes, "TF-IDFC-RF: A Novel Supervised Term Weighting Scheme," *arXiv preprint arXiv:2003.07193*, 2020. DOI: https://doi.org/10.48550/arXiv.2003.07193.

[27] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. DOI: https://doi.org/10.1145/2939672.2939785.

[28] S. Fatima, T. Ghaleb, and L. Briand, *Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests*, 2021. DOI: https://doi.org/10.48550/ARXIV.2112.12331.

[29] Z. Feng, D. Guo, D. Tang, *et al.*, *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*, 2020. DOI: https://doi.org/10.48550/ARXIV.2002.08155.

[30] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021. DOI: https://doi.org/10.1109/ICSE43902.2021.00140.

[31] "What is SAP HANA?" 2022-08-31, archived by Internet Archive at https://web.archive.org/web/20220831125259/https://www.sap.com/products/technology-platform/hana/what-is-sap-hana.html, SAP. (), [Online]. Available: https://www.sap.com/products/technology-platform/hana/what-is-sap-hana.html.

[32] F. Färber, N. May, W. Lehner, *et al.*, "The SAP HANA Database - An Architecture Overview," *IEEE Data Eng. Bull.*, 2012.

[33] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA Database: Data Management for Modern Business Applications," *ACM Sigmod Record*, 2012. DOI: https://doi.org/10.1145/2094114.2094126.

[34] T. Bach, "Testing in Very Large Software Projects," Ph.D. dissertation, Heidelberg University, 2020. [Online]. Available: https://d-nb.info/1259895432/34.

[35] X. Zhu and X. Wu, "Class Noise vs. Attribute Noise: A Quantitative Study," *Artificial intelligence review*, 2004. DOI: https://doi.org/10.1007/s10462-004-0751-8.

[36] G. Pinto, B. Miranda, and M. d'Amorim. "MSR4Flakiness." 2022-09-15, archived by Internet Archive at https://web.archive.org/web/20220915112420/https://github.com/damorimRG/msr4flakiness. (2021), [Online]. Available: https://github.com/damorimRG/msr4flakiness.

[37] "Exploratory Data Analysis, Feature Selection for Better ML Models." 2022-09-07, archived by Internet Archive at https://web.archive.org/save/https://cloud.google.com/blog/products/ai-machine-learning/building-ml-models-with-eda-feature-selection. (2020), [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/building-ml-models-with-eda-feature-selection.

[38] 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. DOI: https://doi.org/10.1145/2635868.2635920.

[39] R. Hudson, "Explicating Exact versus Conceptual Replication," *Erkenntnis*, Sep. 2021. DOI: 10.1007/s10670-021-00464-z.

[40] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know You Neighbor: Fast Static Prediction of Test Flakiness," *IEEE Access*, vol. 9, 2021. DOI: https://doi.org/10.1109/ACCESS.2021.3082424.

[41] B. W. Yap, K. A. Rani, H. A. A. Rahman, S. Fong, Z. Khairudin, and N. N. Abdullah, "An Application of Oversampling, Undersampling, Bagging and Boosting in Handling Imbalanced Datasets," in *Proceedings of the first international conference on advanced data and information engineering (DaEng-2013)*, Springer, 2014.

[42] "RandomUnderSampler Documentation." 2022-11-02, archived by Internet Archive at http://web.archive.org/web/20221103125945/https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html?highlight=randomundersampler. (2022), [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html?highlight=randomundersampler.

[43] "Introduction to Information Retrieval." 2022-10-28, archived by Internet Archive at http://web.archive.org/web/20221028165043/https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html. (2008), [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html.

[44] "Token Documentation." 2022-11-14, archived by Internet Archive at http://web.archive.org/web/20221114114727/https://docs.python.org/3/library/token.html. (2022), [Online]. Available: https://docs.python.org/3/library/token.html.

[45] "NLTK Documentation." 2022-11-14, archived by Internet Archive at http://web.archive.org/web/20221028163019/https://www.nltk.org/api/nltk.tokenize.html. (2022), [Online]. Available: https://www.nltk.org/api/nltk.tokenize.html.

[46] J. Shi, Z. Yang, J. He, B. Xu, and D. Lo, *Can Identifier Splitting Improve Open-Vocabulary Language Model of Code?* 2022. DOI: https://doi.org/10.48550/arXiv.2201.01988.

[47] M. Hucka, "Spiral: Splitters for Indentifiers in Source Code Files," *Journal of Open Source Software*, 2018. DOI: https://doi.org/10.21105/joss.00653.

[48] G. An, J. Yoon, J. Sohn, J. Hong, D. Hwang, and S. Yoo, "Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA," in *Proceedings of the 44th ICSE: SEiP*, 2022. DOI: https://doi.org/10.1145/3510457.3513051.

[49] C. Lee and D. Landgrebe, "Feature Extraction based on Decision Boundaries," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1993. DOI: https://doi.org/10.1109/34.206958.

[50] 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*, 2021. DOI: https://doi.org/10.1109/ICPC52881.2021.00052.

[51] "TF-IDFC-RF GitHub." 2022-11-05, archived by Internet Archive at http://web.archive.org/web/20221105094442/https://github.com/LaCAfe/TF-IDFC-RF. (2022), [Online]. Available: https://github.com/LaCAfe/TF-IDFC-RF.

[52] D. Arp, E. Quiring, F. Pendlebury, *et al.*, *Dos and Don'ts of Machine Learning in Computer Security*, 2020. DOI: https://doi.org/10.48550/ARXIV.2010.09470.

[53] "KFold Scikit-Learn Documentation." 2022-11-14, archived by Internet Archive at http://web.archive.org/web/20221114120922/https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html. (2022), [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html.

[54] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. 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)*, 2021. DOI: https://doi.org/10.1109/MSR52588.2021.00034.

[55] "SAP HANA Virtual Tables Documentation." 2022-11-12, archived by Internet Archive at http://web.archive.org/web/20221112121559/https://help.sap.com/docs/SAP_POWERDESIGNER/5b9c2c4e13a94769bff7cfa6524dac68/524c55a7a8f948e79874647f91a98805.html?version=16.7.04. (2022), [Online]. Available: https://help.sap.com/docs/SAP_POWERDESIGNER/5b9c2c4e13a94769bff7cfa6524dac68/524c55a7a8f948e79874647f91a98805.html?version=16.7.04.

[56] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An Empirical Study of Bugs in Test Code," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, 2015. DOI: https://doi.org/10.1109/ICSM.2015.7332456.

[57] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," 2019. DOI: https://doi.org/10.1145/3338906.3338945.

[58] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A Study on the Lifecycle of Flaky Tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020. DOI: https://doi.org/10.1145/3377811.3381749.