# Test Flakiness Prediction Techniques for Evolving Software Systems


**Presented by:**
Guillaume HABEN, University of Luxembourg, Luxembourg


**Defense committee:**

| | | |
|---|---|---|
| Prof. Dr. Mike Papadakis, | Chairman, | University of Luxembourg, Luxembourg |
| Dr. Maxime Cordy, | Vice-chairman, | University of Luxembourg, Luxembourg |
| Prof. Dr. Yves Le Traon, | Supervisor, | University of Luxembourg, Luxembourg |
| Prof. Dr. Arie Van Deursen, | Member & Reviewer, | TU Delft, The Netherlands |
| Prof. Dr. Javier Tuya, | Member & Reviewer, | Universidad de Oviedo, Spain |

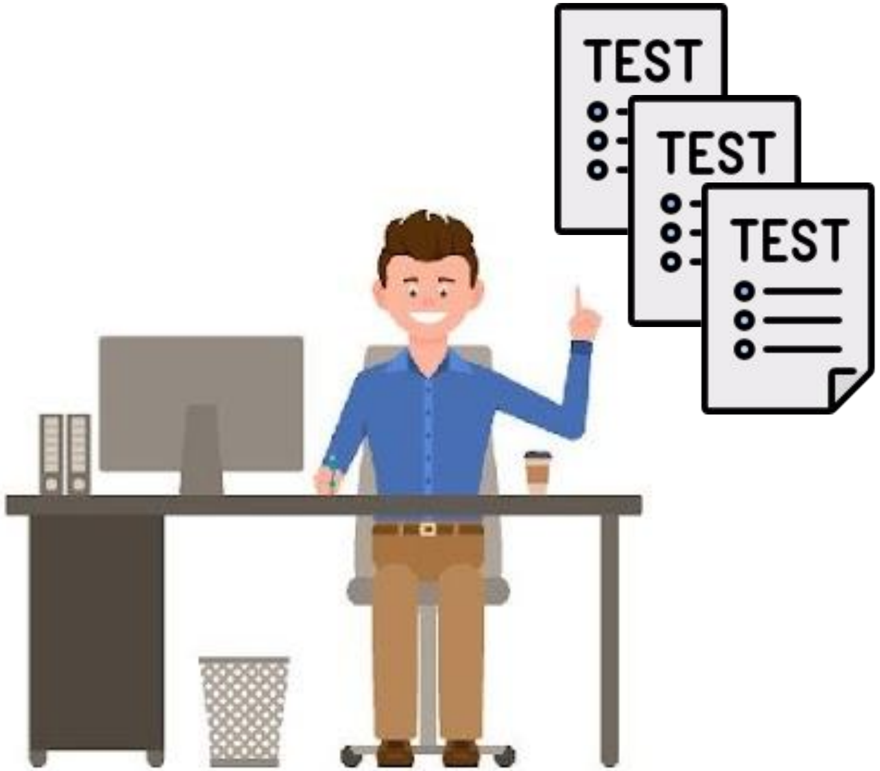
**Date:**
June 29th, 2023

PayPal
Serval
securityandtrust.lu
SnT
securityandtrust.lu
uni.lu
UNIVERSITÉ DU LUXEMBOURG

# Jim

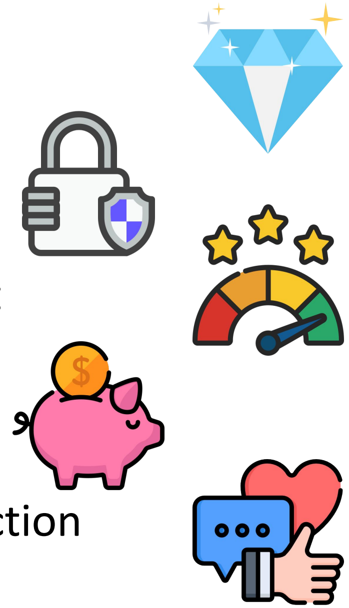
# Software Testing

## Identifying issues and defects before software is released




**Benefits:**

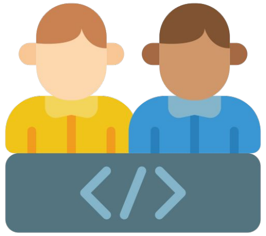
- More reliable
- More secure
- More performant
- Save money
- Customer satisfaction



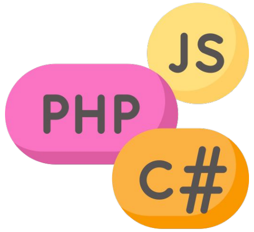

Google

# Google

>10k software engineers



>100m lines of code projects




>1 000 commits per hour

# Challenges

## Testing Large Software Systems

Flakiness

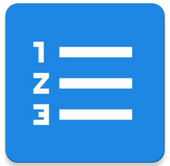
### Handle Large Amounts of Tests

Hundreds of thousands of tests

Avoid Anti-Patterns

Regression Test Selection

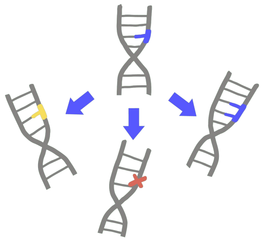
Test Case Prioritization



### Ensure Test Quality

Test Coverage

Test Robustness

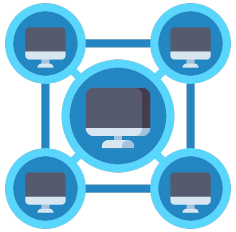


Test Refactoring

### Manage Multi-Environments

Dev, Test, Prod ≠

Distributed Testing



Platform Dependencies

# Definition

**“A flaky test is a test that**

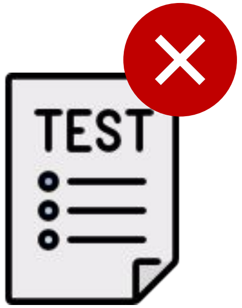
**can both pass or fail when executed several times**

**on the same version of a program”**

# Consequences for developers

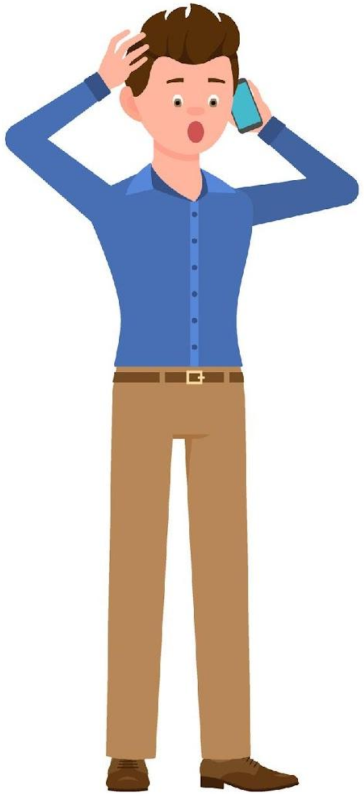
## Gives confusing signals

Real bug?




False alert?


# Investigations



2022 IEEE Conference on Software Testing, Verification and Validation (ICST)

## A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests

Sarra Habchi
University of Luxembourg
sarra.habchi@uni.lu

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

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

Maxime Cordy
University of Luxembourg
maxime.cordy@uni.lu

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

g, Verification and Validation (ICST) | 978-1-6654-6679-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICST53961.2022.00034

***Abstract*—Test flakiness forms a major testing concern. Flaky tests manifest non-deterministic outcomes that cripple continuous integration and lead developers to investigate false alerts. Industrial reports indicate that on a large scale, the accrual of flaky tests breaks the trust in test suites and entails significant computational cost. To alleviate this, practitioners are constrained to identify flaky tests and investigate their impact. To shed light on such mitigation mechanisms, we interview 14 practitioners with the aim to identify (i) the sources of flakiness within the testing ecosystem, (ii) the impacts of flakiness, (iii) the measures adopted by practitioners when addressing flakiness, and (iv) the automation opportunities for these measures. Our analysis shows that, besides the tests and code, flakiness stems from interactions between the system components, the testing infrastructure, and external factors. We also highlight the impact of flakiness on testing practices and product quality and show that the adoption of guidelines together with a stable infrastructure are key measures in mitigating the problem.**

### I. Introduction

Software Testing is critical for modern software development as it allows the concurrent implementation and integration of features. At Google, more than 50 million test cases are executed every day to ensure the quality of their products [1]. Though, test automation faces major problems with the emergence of flaky tests [2]–[4]. Flaky tests are tests that, for the same versions of code and test, can pass and fail on different runs. Such non-determinism sends confusing signals to developers who struggle to interpret the test results. As a result, developers lose trust in test suites, disregard their signals and integrate features containing real failures, thereby nullifying the purpose of testing.

are among the main categories of test flakiness. Notably, the study of Eck *et al.* [7] showed that flakiness can stem from the code under test and highlighted its potential impact on organisational aspects like resource allocation.

Other studies investigated tools and techniques that could help developers to cope with test flakiness. Automated tools, such as DeFlaker [11], iDFlakies [12], and FlakeFlagger [13] have been developed in order to detect flaky tests with a minimum number of test runs or re-runs. Unfortunately, these advances offer only partial solutions to the problem and may not fit well within the development systems and organisation constraints. For instance, DeFlaker relies on coverage and reruns of tests that do not execute changed code, which are not possible in specific development environments that use regression test selection or when coverage cannot be obtained. Furthermore, the fixing of flaky tests gained traction as studies investigated the fixing effort devoted for flaky tests and tools like [14] are designed to fix flaky tests. Nonetheless, in order to devise flakiness solutions, we need to understand how developers deal with flaky tests in practice. In particular, it is necessary to identify the typical measures taken by practitioners when dealing with flaky tests, and reflect on how research solutions could assist and improve them.

To shed some light on these questions, we conduct an empirical study focused on the industrial context in which flakiness manifests. Specifically, we perform a qualitative analysis on data collected from 14 practitioner interviews to answer the following research questions:

- **RQ1:** Where can we locate flakiness?

**Dealing with flakiness:**

**Ignore** → # flaky tests will grow

**Remove** → Lose test information

**Quarantine** → Postponing actions

**Fix** → Rarely achieved

**Reruns** → Go to “solution”

Habchi, Sarra, et al. ”A Qualitative Study on the Sources, Impacts and Mitigation Strategies of Flaky Tests" *Proceedings of the 15th International Conference on Software Testing, Verification and Validation (ICST)*, 2022

# Why does it matter?

Flaky tests often accounts for 1-5%

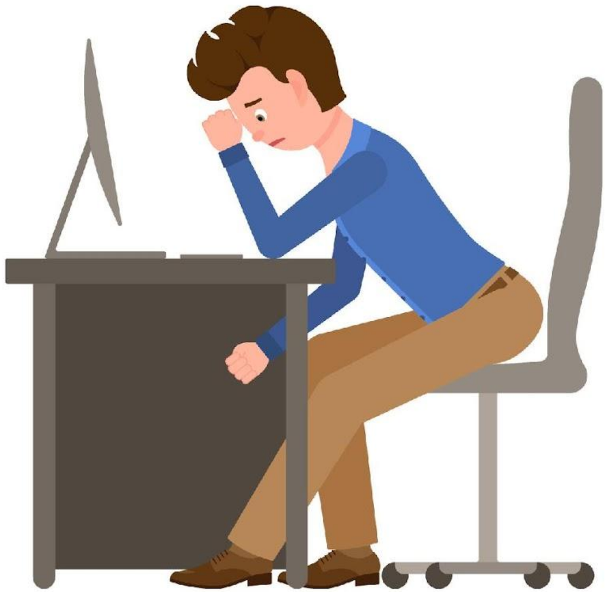
Flakiness increases costs both time-wise and computer-wise

- At Google: up to 16% of testing budget spent just to rerun flaky tests

Flakiness reduces productivity (delay builds) and trust

This leads to bad quality

**Major problem in software testing**



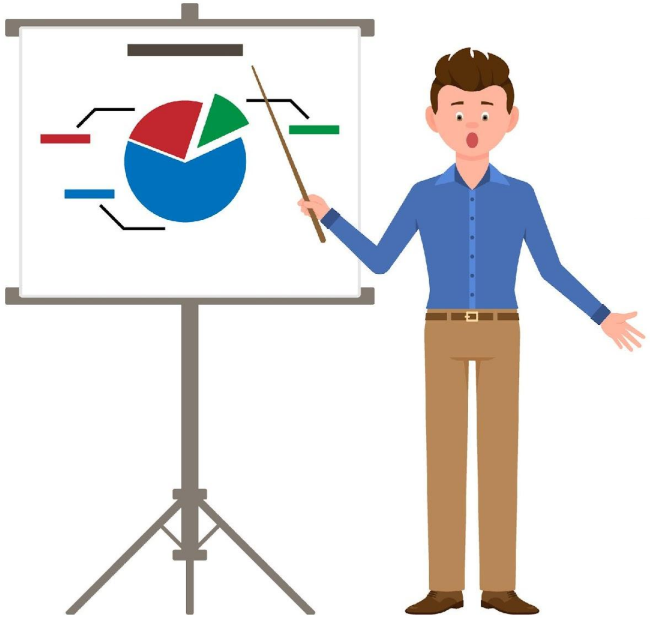
Micco, John and Memon, Atif ”GTAC 2016: How flaky tests in continuous integration”, https://www.youtube.com/watch?v=CrzpkF1-VsA, 2016

# Concrete example of a flaky test

```python
# https://github.com/python-telegram-bot/python-telegram-bot/blob/master/tests/test_updater.py
def test_idle(self, updater, caplog):
       updater.start_polling(0.01)
       Thread(target=partial(self.signal_sender, updater=updater)).start()
       with caplog.at_level(logging.INFO):
           updater.idle()
       rec = caplog.records[-2]
       assert rec.getMessage().startswith('Received signal {signal.SIGTERM}')
       assert rec.levelname == 'INFO'
       rec = caplog.records[-1
       assert rec.getMessage().startswith('Scheduler has been shut down')
       assert rec.levelname == 'INFO'
       # If we get this far, idle() ran through
       sleep(0.5)
       assert updater.running is False
```

# Root cause

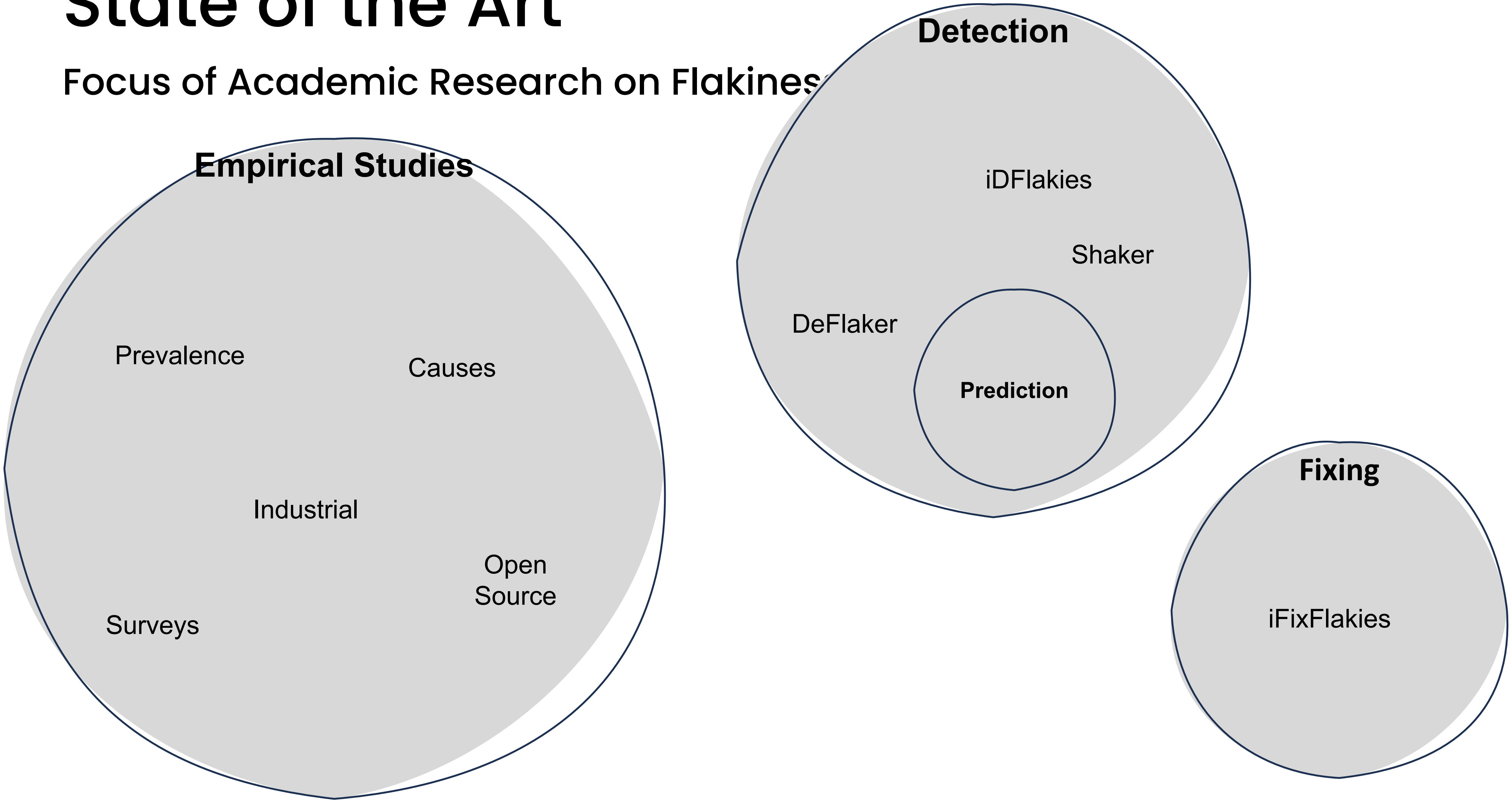
## Categories of flakiness



| Category | Definition | Sources |
|---|---|---|
| Asynchronous Waits | Flakiness caused by tests that involve asynchronous operations and have dependencies on timing, resulting in inconsistent behaviour if the expected response is not received within a specified time. | [49], [58], [61], [62] |
| Concurrency | Flakiness caused by race conditions or synchronisation issues when multiple threads or processes interact with shared resources simultaneously, leading to unpredictable outcomes. | [49], [58], [61], [62] |
| Time | Tests depending on specific timing conditions, such as time-sensitive calculations or time-based events, and may produce different results based on the time of execution. | [49], [58], [61], [62] |
| Order-Dependency | Flakiness resulting from tests that rely on a specific execution order due to shared resources or dependencies, and may fail if the order among the tests is changed. | [49], [58], [61], [62] |
| Randomness | Flakiness caused by tests that involve random or pseudo-random behaviour, where different outcomes may occur on each run, potentially leading to inconsistent results. | [49], [58], [61], [62] |
| Unordered Collections | Flakiness resulting from tests that rely on unordered collections or sets, where the order of elements can vary, causing failures if the expected order is not maintained. | [58], [61], [62] |
| Network | Flakiness caused by network-related issues, such as unreliable connections, timeouts, or network congestion, leading to inconsistent results in tests that interact with remote services. | [49], [58], [61], [62] |
| I/O (Input/Output) | Flakiness resulting from tests that involve reading from or writing to external files, databases, or other I/O operations, where inconsistencies or errors can occur. | [49], [58], [61], [62] |
| Resource Leak | Flakiness caused by tests that do not release system resources properly, resulting in resource exhaustion and inconsistent behaviour when run repeatedly. | [49], [58], [61], [62] |
| Floating Point | Flakiness caused by tests that rely on the results of floating point operations, which can suffer from discrepancies and inaccuracies due to precision limitations, overflows, non-associative addition, and other factors. | [49], [58], [62] |
| Platform Dependency | Flakiness stemming from tests relying on specific functionalities of an operating system, library version, or hardware vendor. These dependencies can result in inconsistent and non-deterministic test failures, especially in cloud-based continuous integration environments where tests are executed on different platforms. | [49], [61] |
| Test Case Timeout | Flakiness caused by tests that specify an upper limit for the test execution duration. Often those tests will fail because the instructions will not complete in time. | [49], [61] |

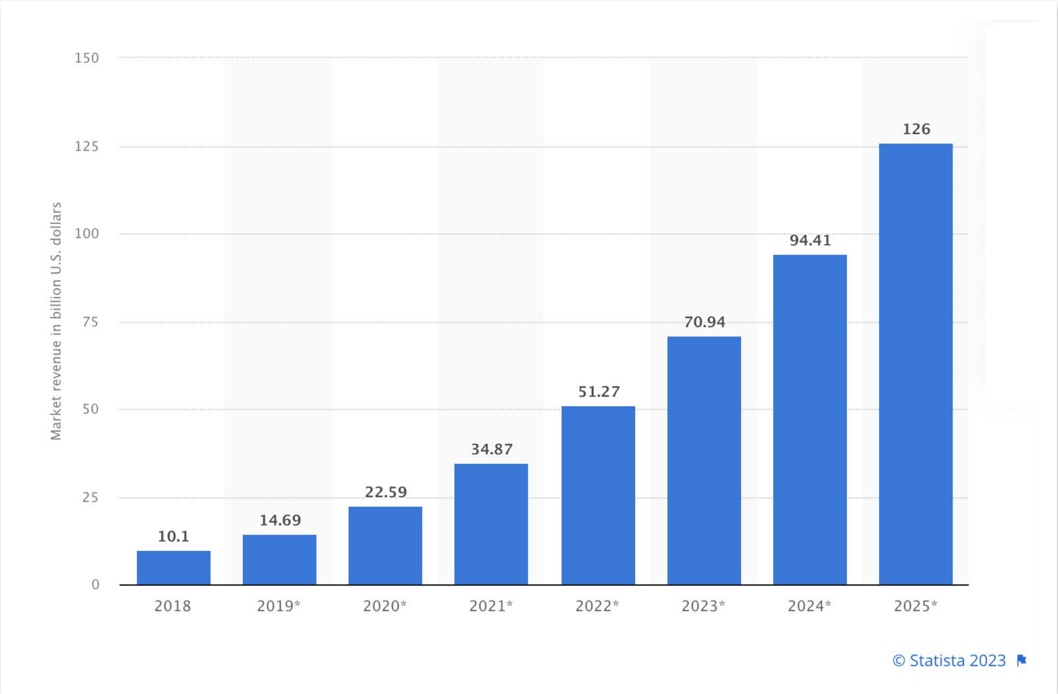
# State of the Art

## Focus of Academic Research on Flakiness



# AI for SE

## The Rise of Artificial Intelligence in the Software Development Industry




Market revenue in billion of $

**Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs**

Anunay Amar and Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University
Montréal, Canada

**Software Defect Prediction via Convolutional Neural Network**

Publisher: IEEE | Cite This | PDF

Jian Li ; Pinjia He ; Jieming Zhu ; Michael R. Lyu   All Authors

**Vulnerability Prediction From Source Code Using Machine Learning**

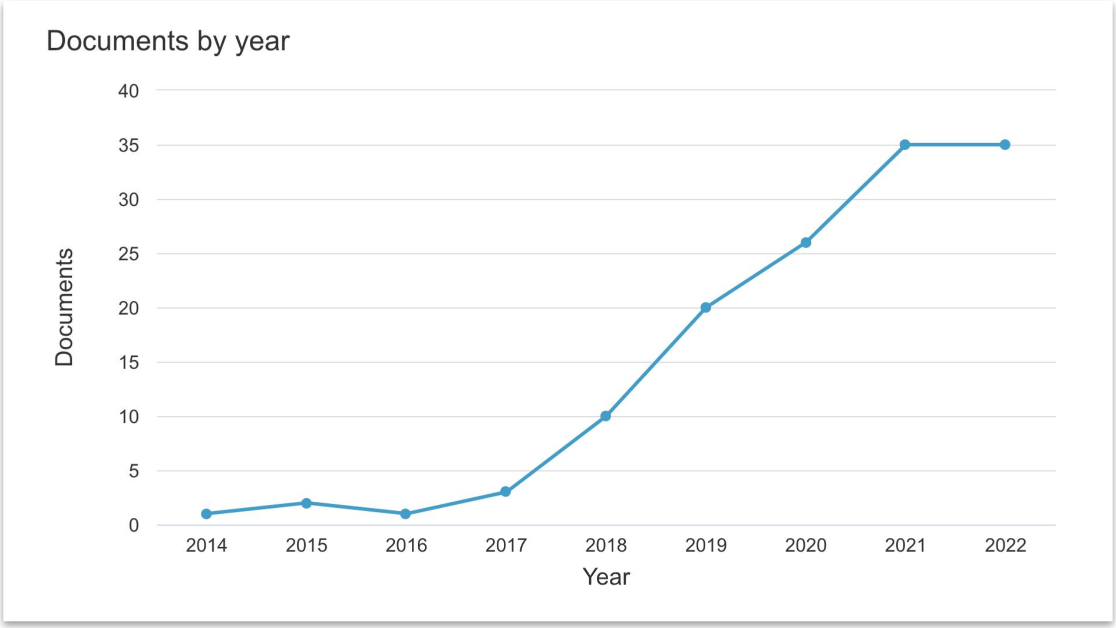
ZEKI BILGIN, (Member, IEEE), MEHMET AKIF ERSOY, ELIF USTUNDAG SOYKAN, EMRAH TOMUR, PINAR ÇOMAK, AND LEYLI KARAÇAY
Ericsson Research, 34367 İstanbul, Turkey

https://www.statista.com/statistics/607716/worldwide-artificial-intelligence-market-revenues/

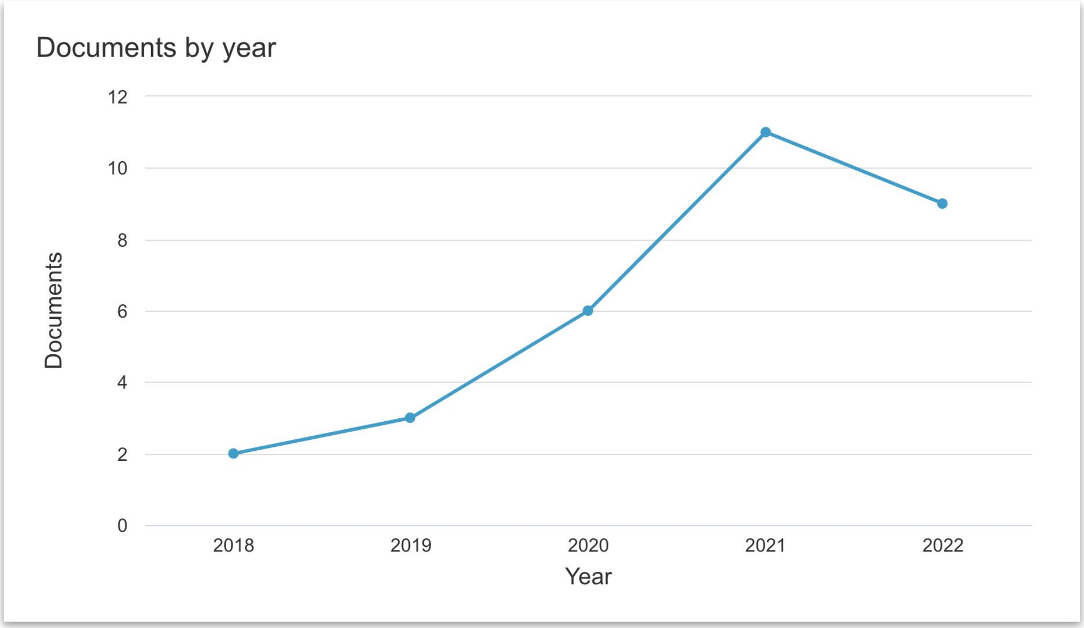
# State of the Art

## Evolution of the Research Interest on Flakiness

Number of published papers mentioning:




“flaky” AND “tests”




“flaky” AND “tests” AND “predict”

https://www.scopus.com/

# Can we also use AI to fight against flakiness?

# Background

## Metrics for binary classification models

| | | Predicted | |
|---|---|---|---|
| | | Negative | Positive |
| Actual | Negative | TN | FP |
| | Positive | FN | TP |

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

$$\textbf{F1} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

$$\textbf{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \in [-1, 1]$$

# Challenges

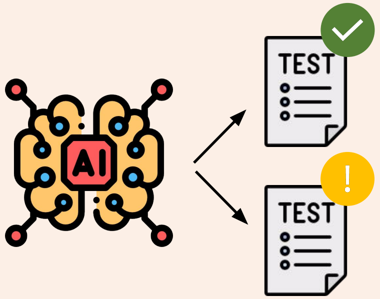
**Challenge #1**
Adoption

- Bridge the gap between Academia and Industry
- More realistic validation
- Concrete case studies

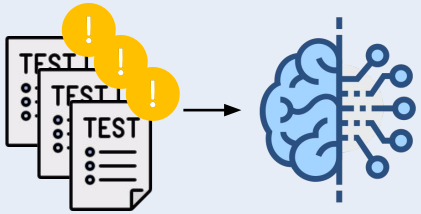
**Challenge #2**
Comprehension


- Understand and locate sources of flakiness
- Better assist developers

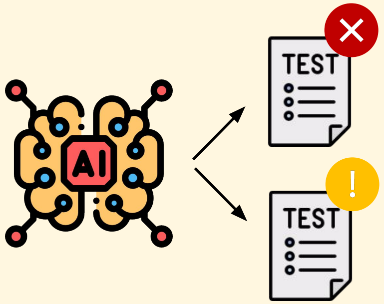
**Question I:**
Can we predict flaky tests?




**Question II:**
Can we predict the category of a flaky test?




**Question IV:**
Are existing prediction techniques suitable to real-world CI?

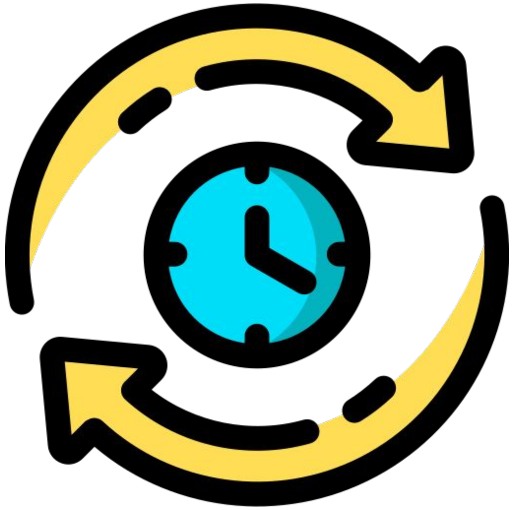


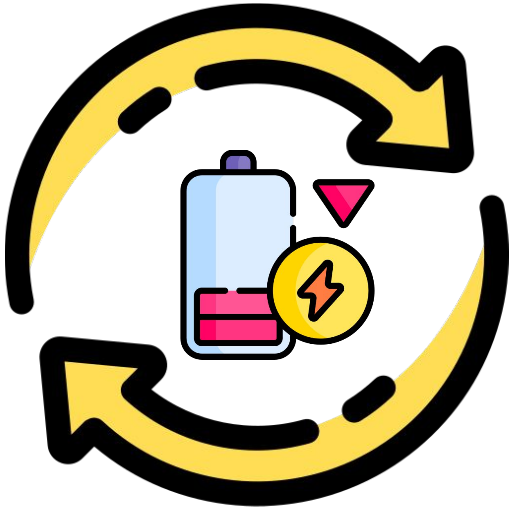

**Question III:**
Can we locate the source of flakiness?

# Contribution #1

# Motivation

Reruns are costly



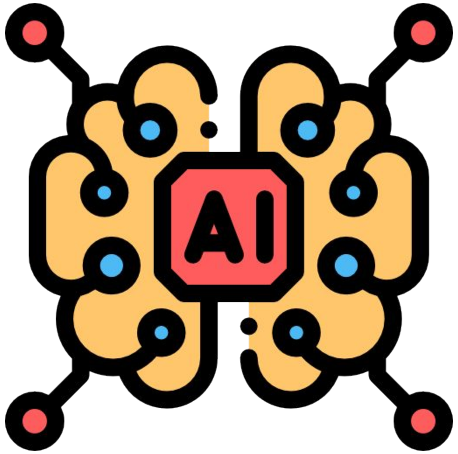


# Motivation

## Released datasets

- DeFlaker (ICSE 2018)
- iDFlakies (ICST 2019)

## Static Prediction




# Vocabulary-based Approach

## Intuition

Idea:
- Relation between certain tokens and flakiness

Advantages:
- Easy to use
- Fast

Representing Test Code

Bag-of-words, n-grams, TF-IDF

→ Counting the occurrences of words/tokens

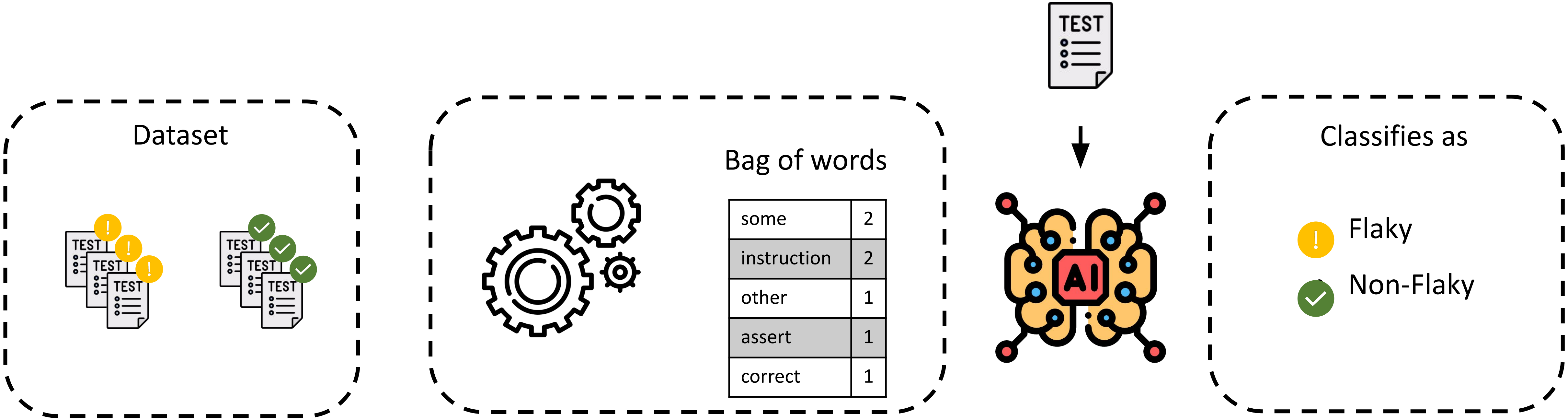
Example

```
@test
exampleTest(param A) {
  some.instruction();
  some.other(instruction);
  assert(correct);
}
```

| | |
|---|---|
| some | 2 |
| instruction | 2 |
| other | 1 |
| assert | 1 |
| correct | 1 |

# Vocabulary-based Approach

## Model overview




Pinto, Gustavo, et al. "What is the vocabulary of flaky tests?" *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020

# Original study by Pinto (MSR 2020)

## Dataset and Results

**Dataset:** DeFlaker
1,348 flaky tests from 6 Java projects

| Algorithm | Precision | Recall | F1 | MCC | AUC |
|---|---|---|---|---|---|
| Random Forest | **0.99** | 0.91 | **0.95** | **0.90** | **0.98** |
| Decision Tree | 0.89 | 0.88 | 0.89 | 0.77 | 0.91 |
| Naive Bayes | 0.93 | 0.80 | 0.86 | 0.74 | 0.93 |
| Support Vector | 0.93 | **0.92** | 0.93 | 0.85 | 0.93 |
| Nearest Neighbour | 0.97 | 0.88 | 0.92 | 0.85 | 0.93 |

Pinto, Gustavo, et al. "What is the vocabulary of flaky tests?." Proceedings of the 17th International Conference on Mining Software Repositories. 2020.
Bell, Jonathan, et al. "DeFlaker: Automatically Detecting Flaky Tests" *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018

## Adoption

# Can we predict flaky tests…

# in more realistic settings?

# Motivation

## The importance of intra-project analysis and time-sensitive validation

inter-project

Original work

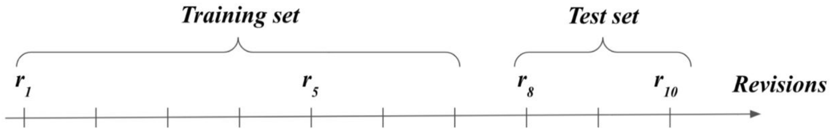
No time-sensitivity

Original work

intra-project

This work

time-sensitivity

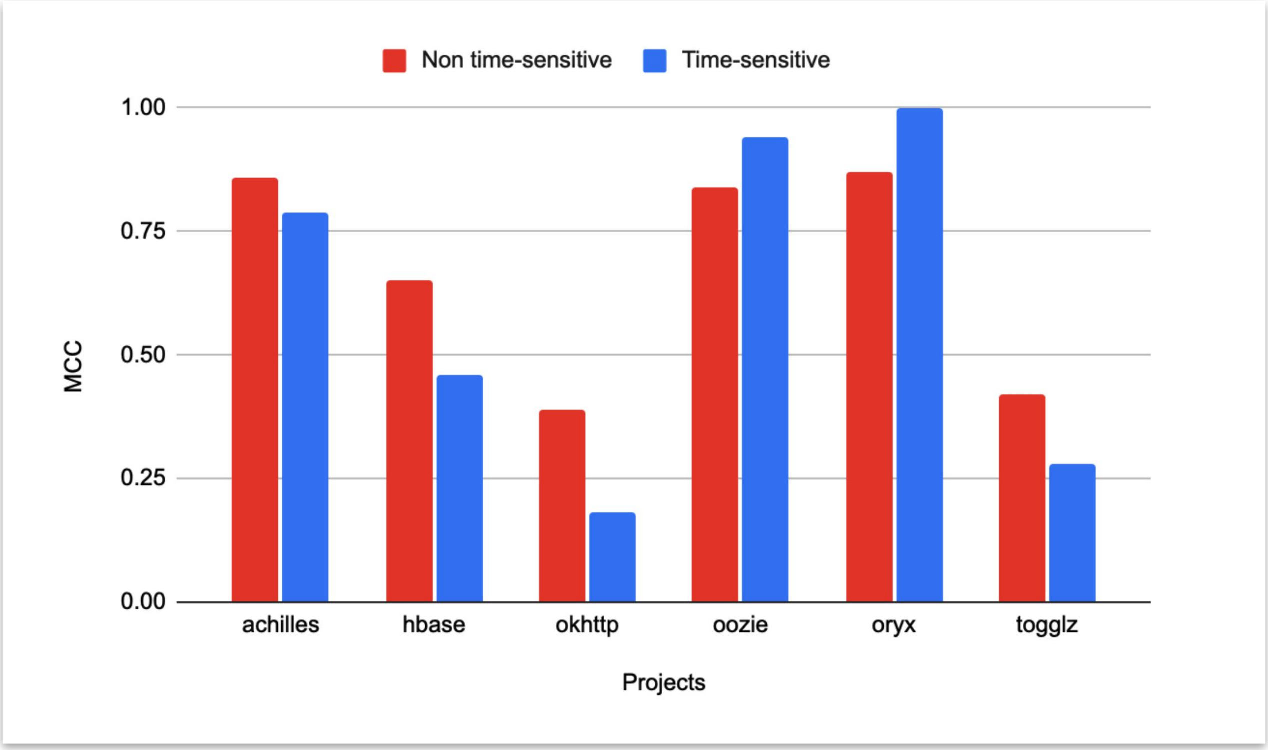



This work

# 1. Per project, in time and non-time sensitive settings

## Classifier (Random Forest) performance

| Project | Earliest revision | latest revision | #FT | #NFT |
|---|---|---|---|---|
| achilles | 2015-10-30 | 2016-09-05 | 51 | 392 |
| hbase | 2010-05-17 | 2010-06-21 | 98 | 120 |
| okhttp | 2014-03-06 | 2015-01-30 | 102 | 1178 |
| oozie | 2013-03-20 | 2013-05-31 | 1039 | 44 |
| oryx | 2015-01-06 | 2015-02-27 | 38 | 286 |
| togglz | 2016-01-23 | 2016-06-17 | 20 | 256 |




Intra-projects and time-sensitive analysis give lower performance, but the prediction of flaky tests is still promising

# 2. Generalisable to other programming languages

## Dataset and Experimental setup

Github Mining

- Python projects
- @flaky annotation

| Project | SHA | #FT | #NFT |
|---|---|---|---|
| bokeh | ddc22b8 | 100 | 2505 |
| cassandra-dtest | 8cb6bd2 | 72 | 4221 |
| celery | 0833a27 | 54 | 2890 |
| jira | 7fa3a45 | 131 | 59 |
| pipenv | 8e64873 | 32 | 1612 |
| python-amazon | 84c16f5 | 35 | 15 |
| python-telegram-bot | 8e7c0d6 | 186 | 1382 |
| spyder | 413c994 | 173 | 1086 |
| typed-python | 96e7ebd | 54 | 6034 |

# 2. Generalisable to other programming languages

## Classifier performance for Python projects

| Project | Precision | Recall | F1 | MCC | AUC |
|---|---|---|---|---|---|
| bokeh | **1.00** | **0.91** | **0.95** | **0.95** | **0.95** |
| cassandra-dtest | **0.96** | 0.43 | 0.58 | 0.63 | 0.71 |
| celery | 0.85 | 0.54 | 0.64 | 0.66 | 0.77 |
| jira | **0.98** | **0.99** | **0.99** | **0.95** | **0.98** |
| pipenv | 0.78 | 0.19 | 0.30 | 0.37 | 0.60 |
| python-amazon | **0.97** | **1.00** | **0.99** | **0.95** | **0.96** |
| python-telegram-bot | **1.00** | **0.99** | **1.00** | **0.99** | **1.00** |
| spyder | **0.92** | 0.77 | 0.83 | 0.82 | 0.88 |
| typed-python | **1.00** | 0.86 | **0.91** | **0.92** | **0.93** |

Vocabulary-based prediction is generalisable to other programming languages

# 3. Predicting manifest flaky tests?

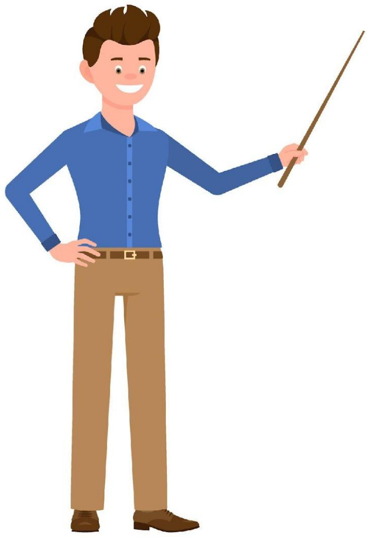
## Experimental setup and results

| Project | #reruns | #@flaky | #manifest FT |
|---|---|---|---|
| bokeh | 200 | 100 | 1 |
| celery | 300 | 54 | 2 |
| python-telegram-bot | 300 | 186 | 20 |

| Project | Precision |
|---|---|
| python-telegram-bot | 1.00 |

Models can help developers mark flaky tests as flaky

# Take-away messages

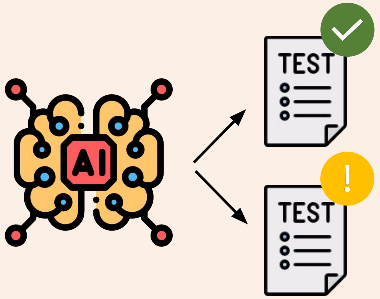


A promissing approach to assit developers

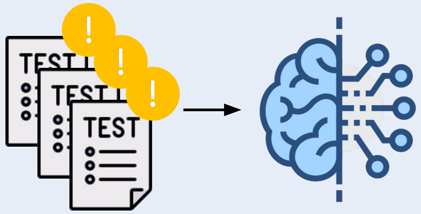
Enough data required to reach good accuracy

Important to validate approaches in realistic settings

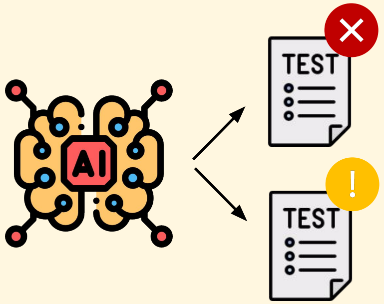
**Question I:**
Can we predict flaky tests?




**Question II:**
Can we predict the category of a flaky test?




**Question IV:**
Are existing prediction techniques suitable to real-world CI?




**Question III:**
Can we locate the source of flakiness?

# Contribution #2

# Real case of a flaky test

```
# https://github.com/python-telegram-bot/python-telegram-bot/blob/master/tests/test_updater.py
def test_idle(self, updater, caplog):
       updater.start_polling( 0.01)
       Thread(target=partial(self.signal_sender, updater=updater)).start()
       with caplog.at_level(logging.INFO):
           updater.idle()
       rec = caplog.records[ -2]
       assert rec.getMessage().startswith( 'Received signal {signal.SIGTERM}')
       assert rec.levelname == 'INFO'
       rec = caplog.records[ -1
       assert rec.getMessage().startswith( 'Scheduler has been shut down' )
       assert rec.levelname == 'INFO'
       # If we get this far, idle() ran through
       sleep(0.5)
       assert updater.running is False
```

Concurrency issue?

Network call/latency?

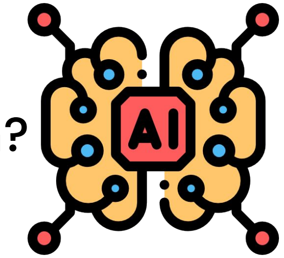
Asynchronous wait?

# Motivation

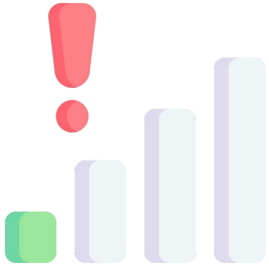
Understanding the cause of a given flaky test remains challenging

Too many flaky tests increase the technical debt, devs need to fix it

Can we use static prediction again?




Problem: How to handle the shortage of data?



# Data Collection

## Dataset

### Existing datasets

- Luo
- Barbosa
- Habchi
- iFixFlakies

### Github mining

### Over-sample Data (SMOTE)

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

Barbosa, Keila, et al. ”Test Flakiness Across Programming Languages" *Transactions on Software Engineering (TSE), 2022*

Habchi, Sarra, et al. ”What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness" *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME)*, 2022

Luo, Qingzhou, et al. ”An Empirical Analysis of Flaky Tests" *Proceedings of the 22th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), 2014*

Shi, August, et al. ”Ifixflakies : A framework for automatically fixing order-dependent flaky tests" *Proceedings of the 27th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), 2019*

# FlakyCat: Siamese networks + Few-Shot Learning

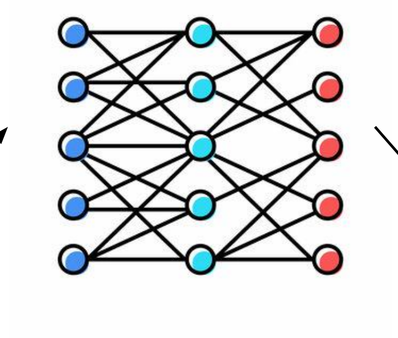
## Intuition

Pros:

- Semantic-aware
- More robust to class imbalance
- Generalize even with few labeled samples

# FlakyCat: Siamese networks + Few-Shot Learning

## Model Overview

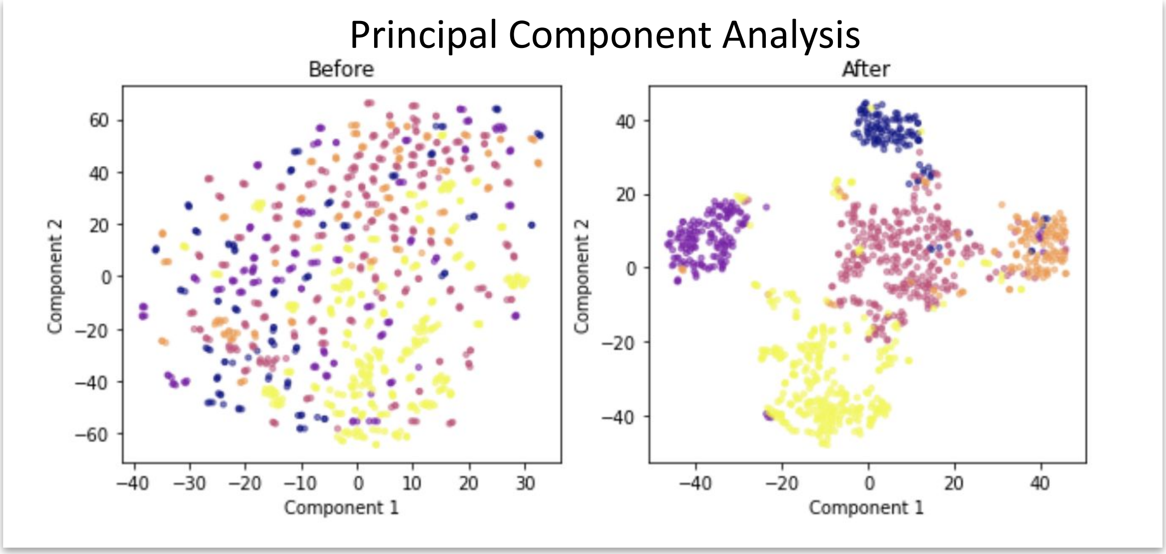


Similarity

Few-shot Learning Classifier

### Category

- Async wait
- Concurrency
- Unordered Collection
- Test Order Dependency
- Time



# Existing Code Representation Techniques

## Vocabulary

```
@test
exampleTest(param A) {
  some.instruction();
  some.other(instruction);
  assert(correct);
}
```

| | |
|---|---|
| some | 2 |
| instruction | 2 |
| other | 1 |
| assert | 1 |
| correct | 1 |

## Test smells

Examples:
- Assertion roulette
- Eager test
- Resource optimist
- …

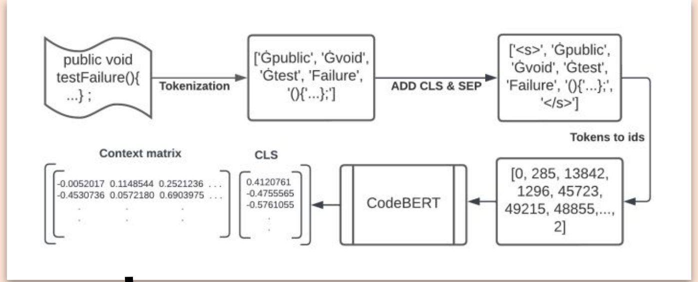
```
@test
exampleTest(param A) {
  some.instruction();
  wait.finish();
  assert(correct);
}
```

Sleepy test

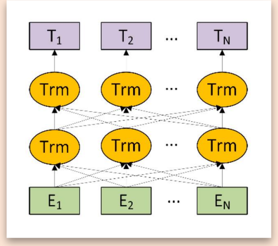
## CodeBERT

Large Language Models

Pre-trained on 6 Programming Languages




Vector representation



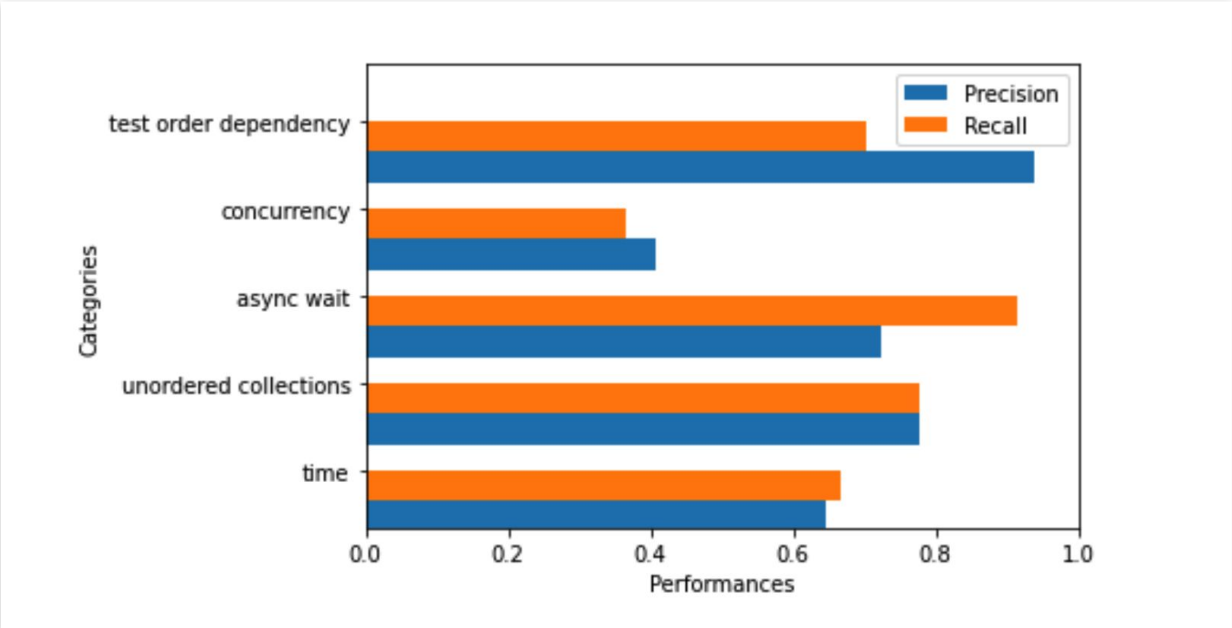
# 1. Combinations of Code Representation and Model Type

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

**FlakyCat**

Flaky Categories prediction is possible despite little data available

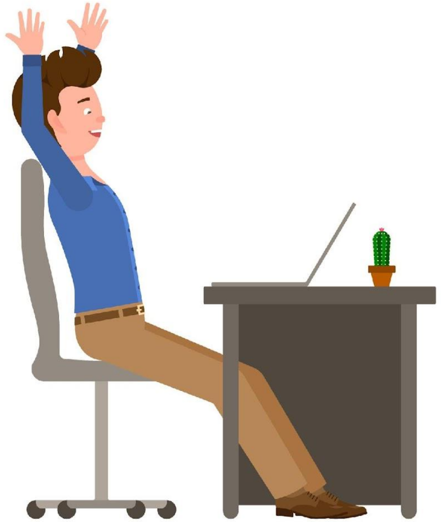
# 2. FlakyCat performance per category




Performance varies depending on the categories

# Take-away messages

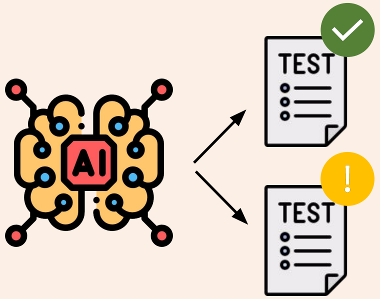
Best performance is achieved using Siamese networks and CodeBERT



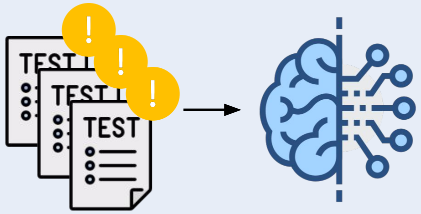
FlakyCat can predict flaky categories

Challenges remain to accurately predict some categories

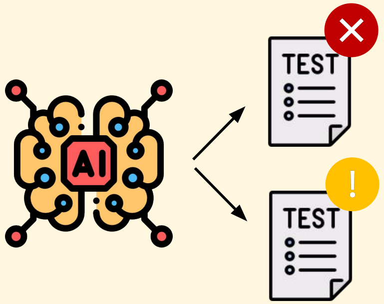
**<u>Question I</u>:**
Can we predict flaky tests?




**<u>Question II</u>:**
Can we predict the category of a flaky test?




**<u>Question IV</u>:**
Are existing prediction techniques suitable to real-world CI?




**<u>Question III</u>:**
Can we locate the source of flakiness?

# Contribution #3

# Motivation

## Little research on fixing flakiness, often limited to one category

- ODRepair & iFixFlakies: Fix order dependencies
- Flex: Fix randomness

## Flakiness sometimes originates from within the program

*"Interestingly, not all flaky tests in this category origin in test code: indeed, the developers report that in 34% of the cases the fixing process requires the examination of the production code and not of the test. Thus, test flakiness can be originated by the production code"*

*"Some fixes to flaky tests (24%) modify the CUT, and most of these cases (94%) fix a bug in the CUT."*

**Help devs to locate flakiness when it originates from the program**

Luo, Qingzhou, et al. "An Empirical Analysis of Flaky Tests" *Proceedings of the 22th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), 2014*

Eck, Moritz, et al. "Understanding Flaky Tests: The Developer's Perspective" *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), 2019*

Comprehension


Can we use Fault Localisation techniques

to find flaky components?

# Fault localisation

- To find bugs/faults
- Most effective technique
- Spectrum-Based Fault Localisation (SBFL)
  → Relies on test coverage and test outcome

# Background

## Regular Spectrum-Based Fault Localisation

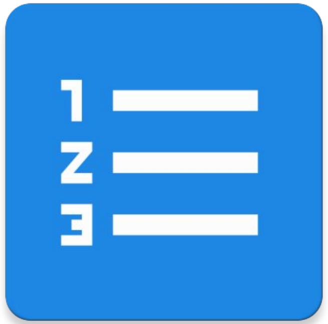
Goal: Finding faulty components based on coverage information

| Line | Program | Tests | | Susp. score |
|---|---|---|---|---|
| | | t1 | t2 | |
| 1 | `int maximum(int a, int b) {` | ■ | ■ | 0.5 |
| 2 | `    if (a > b) {` | ■ | ■ | 0.5 |
| 3 | `        return b; //Fix: return a;` | ■ | □ | 1 |
| 4 | `    }` | ■ | □ | 1 |
| 5 | `    else {` | □ | ■ | 0 |
| 6 | `        return b;` | □ | ■ | 0 |
| 7 | `    }` | □ | ■ | 0 |
| 8 | `}` | ✘ | ✔ | |

SBFL
Example

Output:
Ranked list of elements to inspect.

- 1st stmt: most suspicious
- Last stmt: least suspicious



# Our approach

## Spectrum-Based Flakiness Localisation

TABLE II: SBFL formulae adapted to flakiness.

| **Name** | **Formula** |
|---|---|
| Ochiai [42] | $\frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_s)}}$ |
| Barinel [43] | $1 - \frac{e_s}{e_s+e_f}$ |
| Tarantula [44], [45] | $\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_s}{e_s+n_s}}$ |
| DStar [34] | $\frac{e_f^*}{e_s*n_f}$ |

- Instead of Failing tests and Passing tests
  Using Flaky tests and Stable tests
- Focus on ranking classes

# Data Collection

## Approach

Looking for flakiness-fixing commits: flaky tests with corresponding “flaky” class

1. **Search:** Look for commits in large Java projects containing *flaky* keyword
2. **Inspect:** Limit to atomic commits fixing CUT (*fix, repair, patch* keywords)
3. **Coverage:** Build, run the test suite and get the coverage matrix for all tests
4. **Extract:** Flaky test, “flaky” class, coverages information, cause of flakiness

# Data Collection

## Dataset

TABLE I: Collected Data. *ffc:* number of flakiness-fixing commits. *all:* number of commits in the project.

| **Proj.** | **#Commits** | | **#Tests** | | **#Classes** | |
|---|---|---|---|---|---|---|
| | ffc | all | min - max | avg | min – max | avg |
| Hbase | 8 | 18,990 | 138 - 2,089 | 1,113 | 734 – 1366 | 1053.4 |
| Ignite | 14 | 27,903 | 15 - 1,018 | 174 | 72 – 1767 | 1262.3 |
| Pulsar | 10 | 8,516 | 194 - 1,326 | 626 | 171 – 422 | 259.7 |
| Alluxio | 3 | 32,560 | 315 - 694 | 473 | 131 – 817 | 360.3 |
| Neo4j | 3 | 71,824 | 21 - 5,782 | 2,139 | 40 – 1663 | 581.3 |
| Total | 38 | | 15 - 5,782 | 905 | 40 – 1767 | 820.2 |

# Evaluation Metrics

acc@n — Accuracy: Number of flaky classes ranked in the top n

wef — Wasted Effort: Number of classes inspected before reaching the flaky class

Rwef — Relative effort : Effort wrt the number of covered classes [0, 100]

# Can we use SBFL to identify flaky classes?

| **Project** | **Total** | **acc** | | | | **wef ($\mathbf{R}_{wef}$)** | |
|---|---|---|---|---|---|---|---|
| | | @1 | @3 | @5 | @10 | mean | med |
| Hbase | 8 | 1 | 4 | 5 | 5 | 13.12 (16) | 2.5 (5) |
| Ignite | 14 | 0 | 3 | 3 | 5 | 214.93 (21) | 20.0 (4) |
| Pulsar | 10 | 3 | 5 | 6 | 9 | 9.20 (23) | 3.0 (9) |
| Alluxio | 3 | 0 | 0 | 0 | 1 | 101.67 (65) | 86.0 (83) |
| Neo4j | 3 | 1 | 2 | 2 | 2 | 23.33 (43) | 1.0 (18) |
| Total | 38 | 5 | 14 | 16 | 22 | 94.24 (26) | 6.5 (8) |
| Percentage (%) | 100 | **13** | **37** | 42 | **58** | - | - |

# Can we improve the initial performance?

## Considering other metrics

Flakiness metrics

Change metrics

Size metrics

TABLE III: Code and change metrics used to augment SBFL.

| | Metric | Definition |
|---|---|---|
| Flakiness | #TOPS | Number of time operations performed by the class. |
| | #ROPS | Number of calls to the `random()` method in the class. |
| | #IOPS | Number of input/output operations performed by the class. |
| | #UOPS | Number of operations performed on unordered collections by the class. |
| | #AOPS | Number of asynchronous waits in the class. |
| | #COPS | Number of concurrent calls in the class. |
| | #NOPS | Number of network calls in the class. |
| Change | Changes | Number of unique changes made on the class. |
| | Age | Time interval to the last changes made on the class. |
| | Developers | Number of developers contributing to the class. |
| Size | LOC | The number of lines of code. |
| | CC | Cyclomatic complexity. |
| | DOI | Depth of inheritance. |

# Can we improve this initial performance?

TABLE VI: RQ2: The contribution of flakiness, change, and size metrics to the identification of flaky classes.

| Proj. (#) | SBFL & flakiness | | | | | | SBFL & change | | | | | | SBFL & size | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | acc | | | | wef ($R_{wef}$) | | acc | | | | wef ($R_{wef}$) | | acc | | | | wef ($R_{wef}$) | |
| | @1 | @3 | @5 | @10 | mean | med | @1 | @3 | @5 | @10 | mean | med | @1 | @3 | @5 | @10 | mean | med |
| Hbase (8) | 1 | 4 | 5 | 5 | 11.9 (12) | 3 (4) | 2 | 4 | 4 | 5 | 16.9 (13) | 4 (4) | 2 | 4 | 5 | 5 | 11.4 (12) | 3 (3) |
| Ignite (14) | 0 | 2 | 2 | 4 | 230.9 (26) | 63 (4) | 2 | 4 | 4 | 4 | 222.3 (24) | 18 (4) | 1 | 3 | 3 | 5 | 220.1 (24) | 43 (4) |
| Pulsar (10) | 2 | 5 | 6 | 8 | 10.2 (15) | 3 (8) | 3 | 5 | 7 | 9 | 8.0 (12) | 2 (5) | 2 | 5 | 7 | 9 | 6.9 (13) | 2 (6) |
| Alluxio (3) | 0 | 0 | 1 | 1 | 97.7 (51) | 73 (65) | 0 | 0 | 1 | 1 | 75.7 (49) | 94 (39) | 0 | 0 | 1 | 1 | 90.7 (49) | 77 (58) |
| Neo4j (3) | 1 | 2 | 2 | 2 | 19.3 (42) | 1 (18) | 2 | 2 | 2 | 2 | 6.7 (37) | 0 (9) | 2 | 2 | 2 | 2 | 23.0 (40) | 0 (10) |
| Total (38) | 4 | 13 | 16 | 20 | 99.5 (24) | 8 (8) | 9 | 15 | 18 | 21 | 94.1 (21) | 5 (6) | 7 | 14 | 18 | 22 | 94.3 (22) | 5 (7) |
| Percentage (%) | **11** | 34 | 42 | **53** | - | - | **24** | 39 | 47 | 55 | - | - | **18** | 37 | 47 | 58 | - | - |

| | | | |
|---|---|---|---|
| 5 | 14 | 16 | 22 |
| **13** | **37** | 42 | **58** |

Initial performance

Change and Size metrics have positive impacts

# SBFL + Change + Size metrics

| Project | Total | acc | | | | wef ($R_{wef}$) | |
|---|---|---|---|---|---|---|---|
| | | @1 | @3 | @5 | @10 | mean | med |
| Hbase | 8 | 3 | 5 | 6 | 6 | 9.62 (12) | 1.5 (2) |
| Ignite | 14 | 2 | 4 | 4 | 4 | 228.61 (24) | 17.5 (4) |
| Pulsar | 10 | 3 | 6 | 7 | 9 | 7.30 (12) | 2.0 (5) |
| Alluxio | 3 | 1 | 1 | 1 | 2 | 61.83 (22) | 9.0 (10) |
| Neo4j | 3 | 1 | 2 | 2 | 2 | 19.67 (42) | 1.0 (18) |
| Total | 38 | 10 | 18 | 20 | 23 | 94.61 (19) | 3.5 (5) |
| Perc (%) | 100 | 26 | **47** | 53 | 61 | - | - |

Almost 50% of classes responsible for flakiness are ranked in the top 3

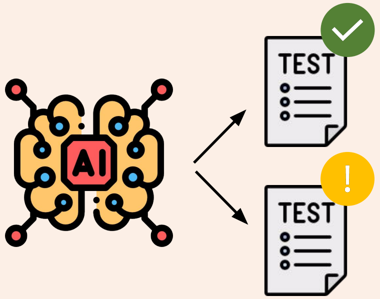
# Take-away messages

We can leverage SBFL to find components in the code causing flakiness

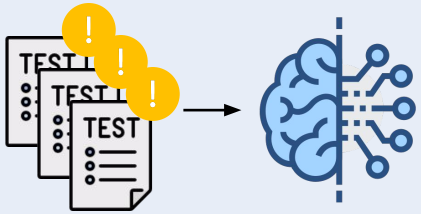
Together, SBFL, change and size metrics give the best results

We need to further help developers

**Question I:**
Can we predict flaky tests?

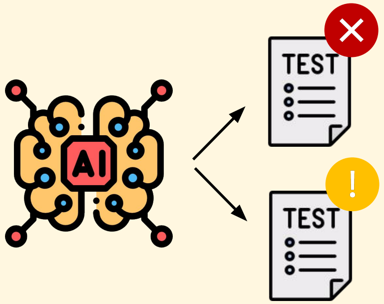



**Question II:**
Can we predict the category of a flaky test?




**Question III:**
Can we locate the source of flakiness?

**Question IV:**
Are existing prediction techniques suitable to real-world CI?



# Contribution #4

# Motivation

## Current Research on Flakiness Prediction

| Study | Model | Feature category | Features | Benchmark | Target | Year |
|---|---|---|---|---|---|---|
| King et al. [91] | Bayesian network | Static & dynamic | Code metrics | Industrial | **Flaky tests** | 2018 |
| Pinto et al. [92] | Random forest | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2020 |
| Bertolino et al. [93] | KNN | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2020 |
| Haben et al. [94] | Random forest | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2021 |
| Camara et al. [95] | Random forest | Static | **Vocabulary** | iDFlakies | **Flaky tests** | 2021 |
| Alshammari et al. [96] | Random forest | Static & dynamic | Code metrics & Smells | FlakeFlagger | **Flaky tests** | 2021 |
| Fatima et al. [97] | Neural Network | Static | CodeBERT | FlakeFlagger iDFlakies | **Flaky tests** | 2021 |
| Pontillo et al. [98] | Logistic regression | Static | Code metrics & Smells | iDFlakies | **Flaky tests** | 2021 |
| Lampel et al. [99] | XGBoost | Static & dynamic | Job execution metrics | Industrial | Flaky failures | 2021 |
| Qin et al. [100] | Neural Network | Static | Dependency graph | Industrial | **Flaky tests** | 2022 |
| Olewicki et al. [101] | XGBoost | Static | **Vocabulary** | Industrial | Flaky builds | 2022 |
| Ackli et al. [102] | Siamese Networks | Static | CodeBERT | Various | **Flaky tests** | 2022 |

Most of the previous research focuses on predicting flaky tests using vocabulary features

# Case study: Chromium

Large project with its own custom CI Framework:

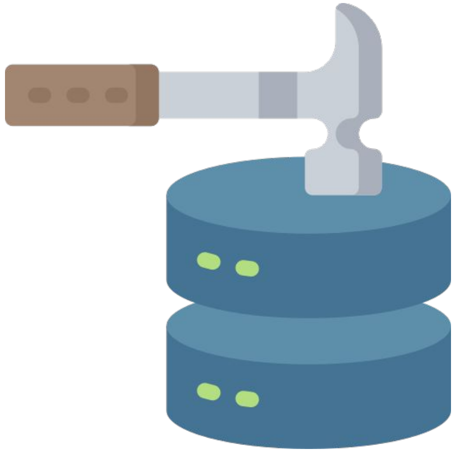
~80 million LOC

Built for hundreds of OS and versions

# Definitions

## Builds

Either:



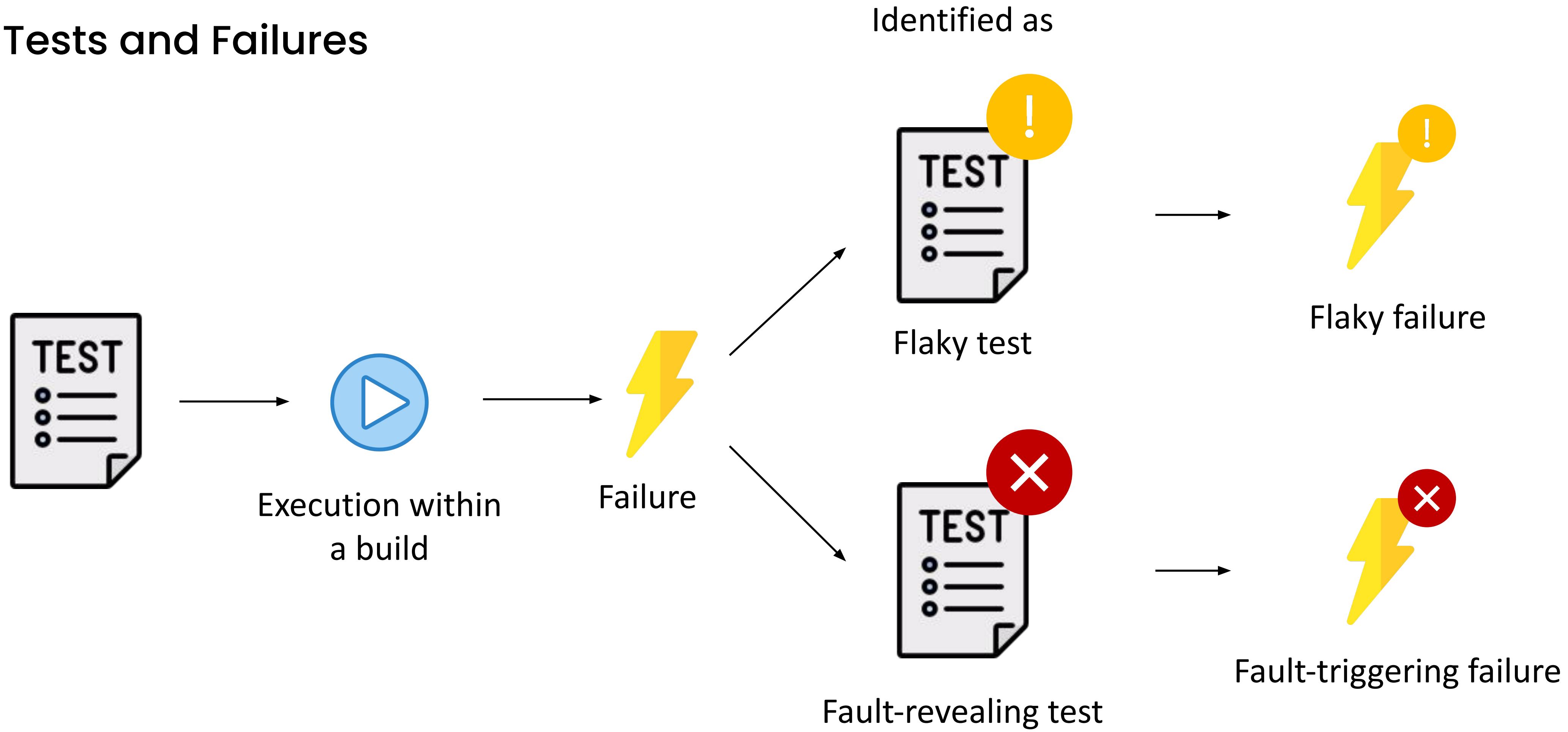
1 build: specific revision

**Builder: Compiles the project**
- Specific version, instrumentations, OS

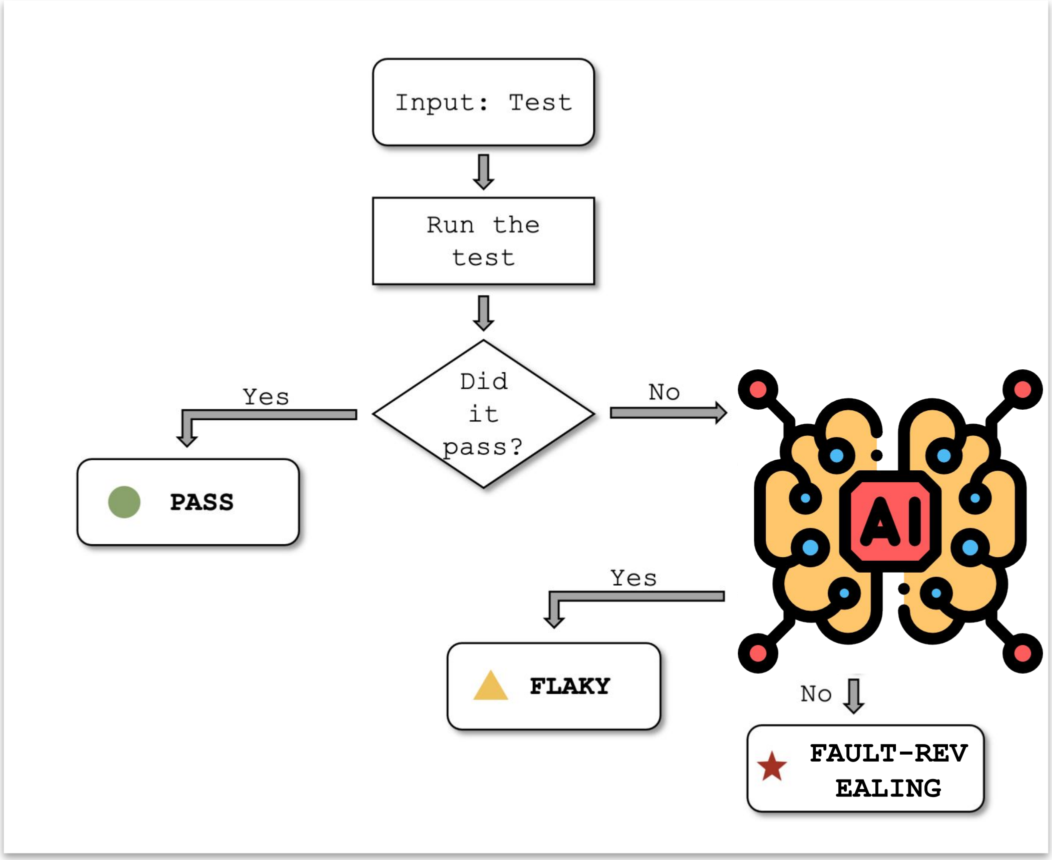
**Tester: Runs regression tests**
~200,000 tests (unit, integration, GUI)

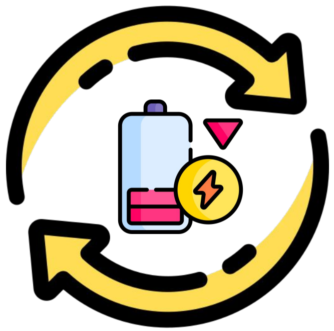
# Definitions

## Tests and Failures



# Identifying flaky tests

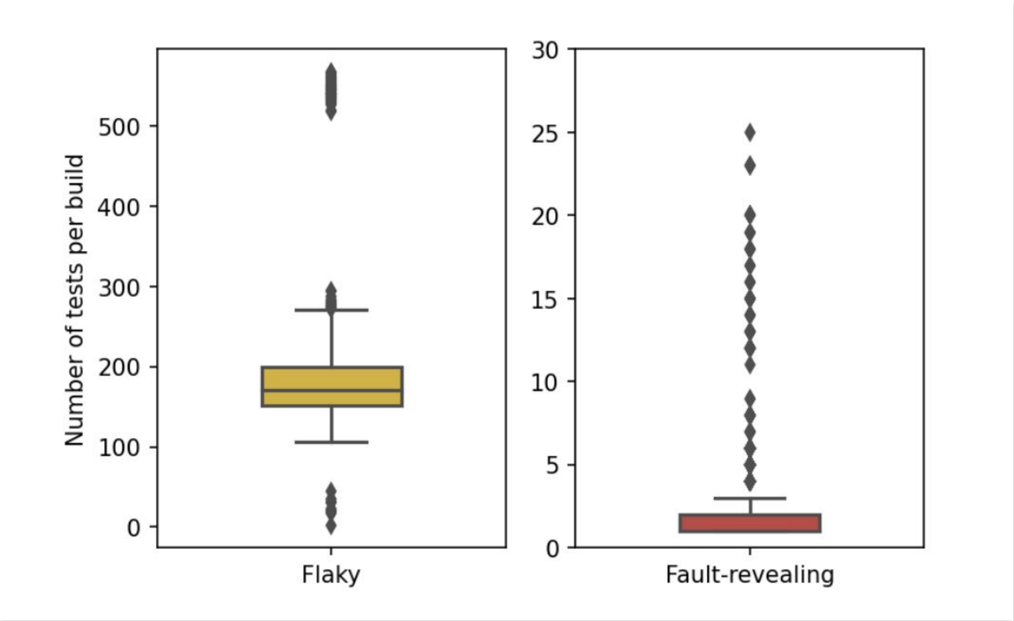


# Data collection

## Dataset

| Tester | Nb of Builds | Period of Time | | Number of Tests | | | Number of Failures | |
|---|---|---|---|---|---|---|---|---|
| | | From | To | Passing | Flaky | Fault-revealing | Flaky | Fault-triggering |
| Linux Tests | 10,000 | Mar 2, 2022 | Dec 1, 2022 | 198,273 | 23,374 | 2,343 | 1,833,831 | 17,171 |




Number of flaky and fault-revealing test per builds

# Facts, Intuition and Questions

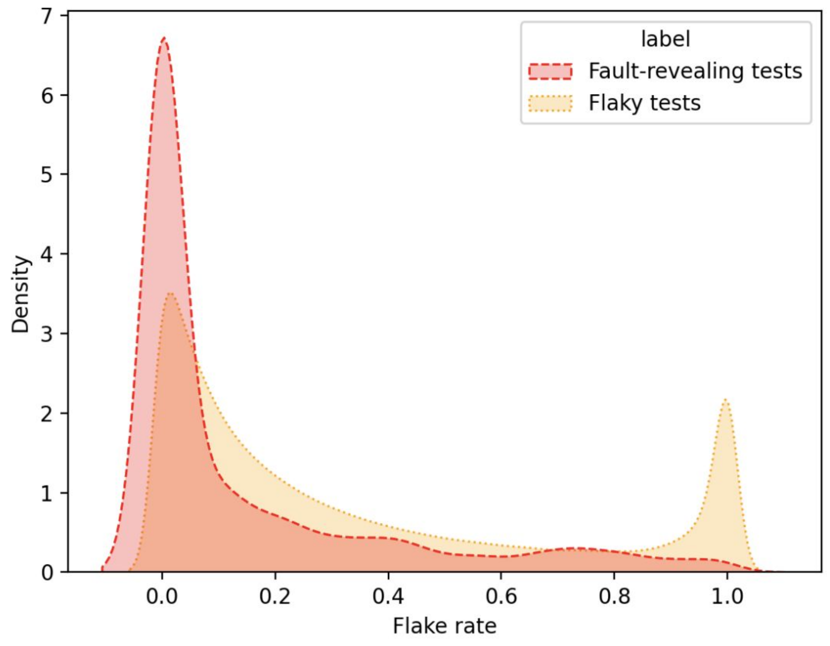
- Fault-revealing tests are blocking, and require investigations
- Costs come from reruns. Reruns occur when there is (at least) one test failure
- Flaky tests are failing because of contextual conditions present during one of their executions

Can we predict failures as flaky or fault-triggering?

# Retrieved features

| Feature Name | Feature Description |
|---|---|
| buildId | The build number associated with the test execution |
| flakeRate | The flake rate of the test over the last 35 builds |
| runDuration | The time spent for this test execution |
| runStatus | ABORT<br>FAIL<br>PASS<br>CRASH<br>SKIP |
| runTagStatus | CRASH<br>PASS<br>FAIL<br>TIMEOUT<br>SUCCESS<br>FAILURE<br>FAILURE_ON_EXIT<br>NOTRUN<br>SKIP<br>UNKNOWN |
| testSource | The test source code |
| testSuite | The test suite the test belongs to |
| testId | The test name |

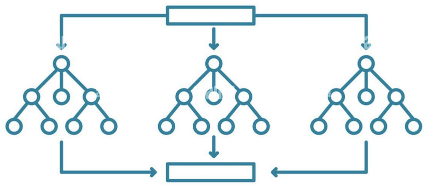
The flake rate is often used in the industry to quantify the level of flakiness of a test



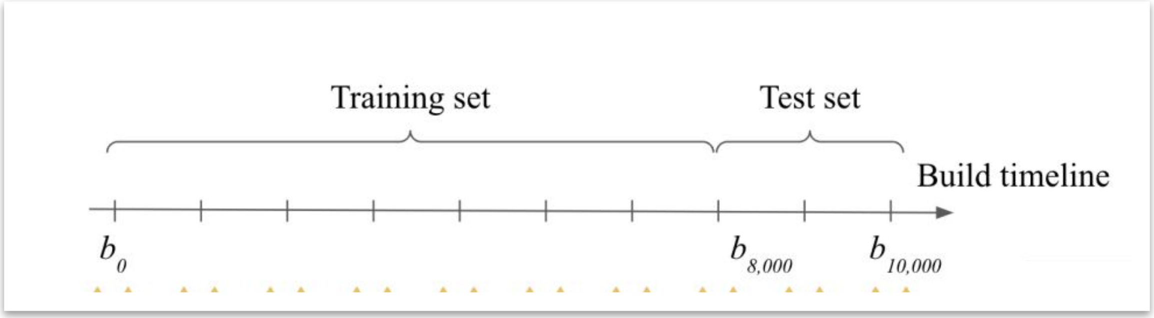
# Experimental settings

## Model training

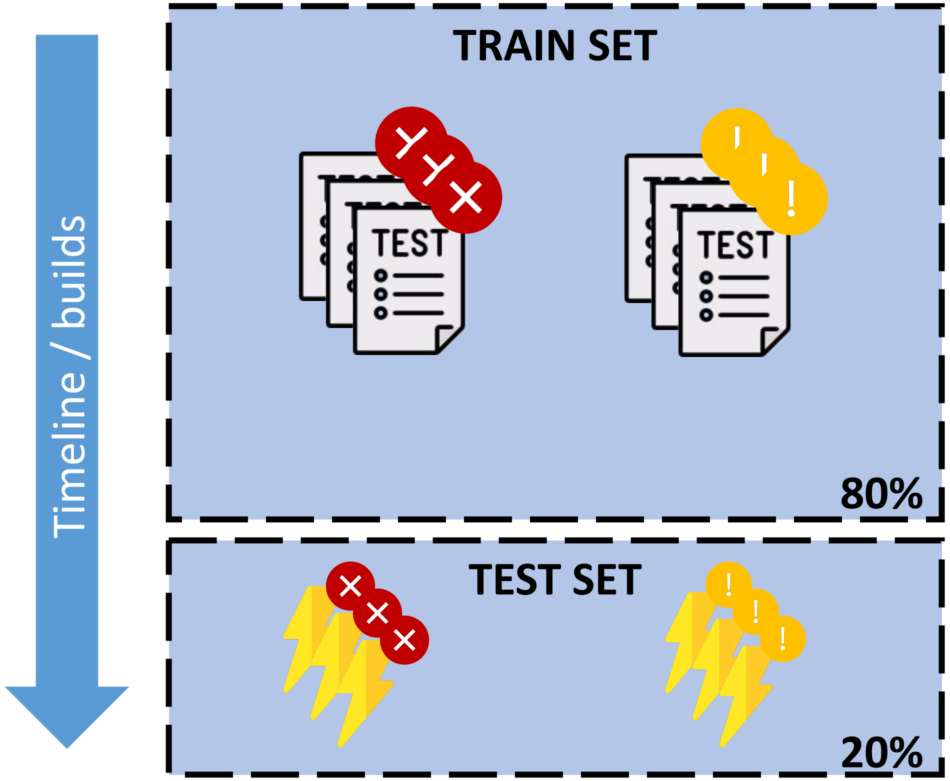
Random Forest Classifier



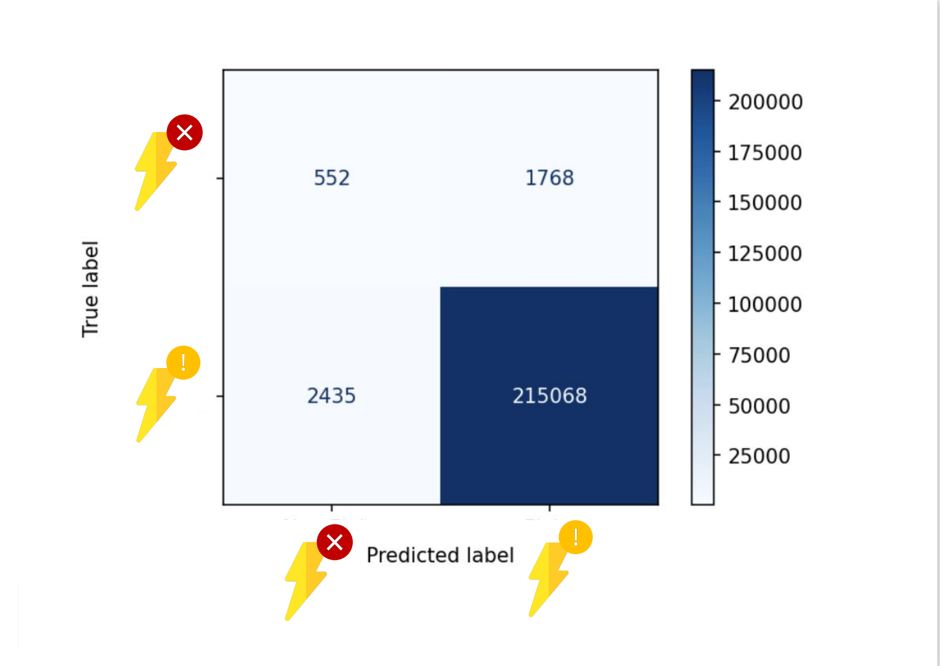
Time-sensitive analysis



# 1. Performance of existing approaches

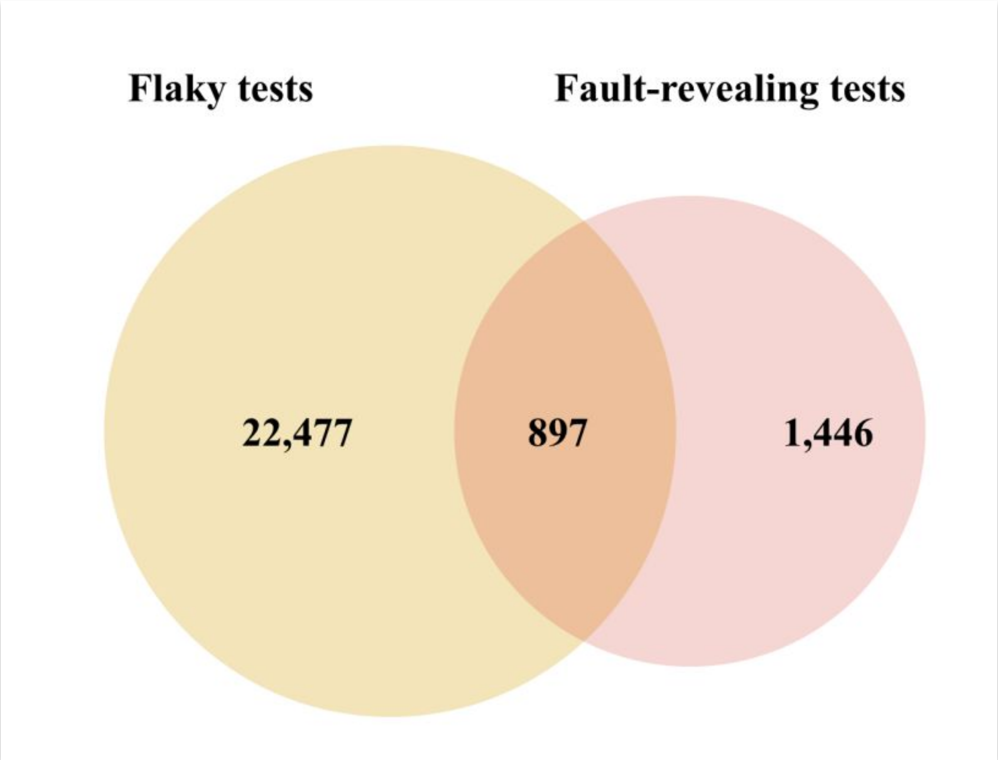



| Precision | Recall | MCC | FPR |
|---|---|---|---|
| 99.2% | 98.9% | 0.20 | 76.2% |



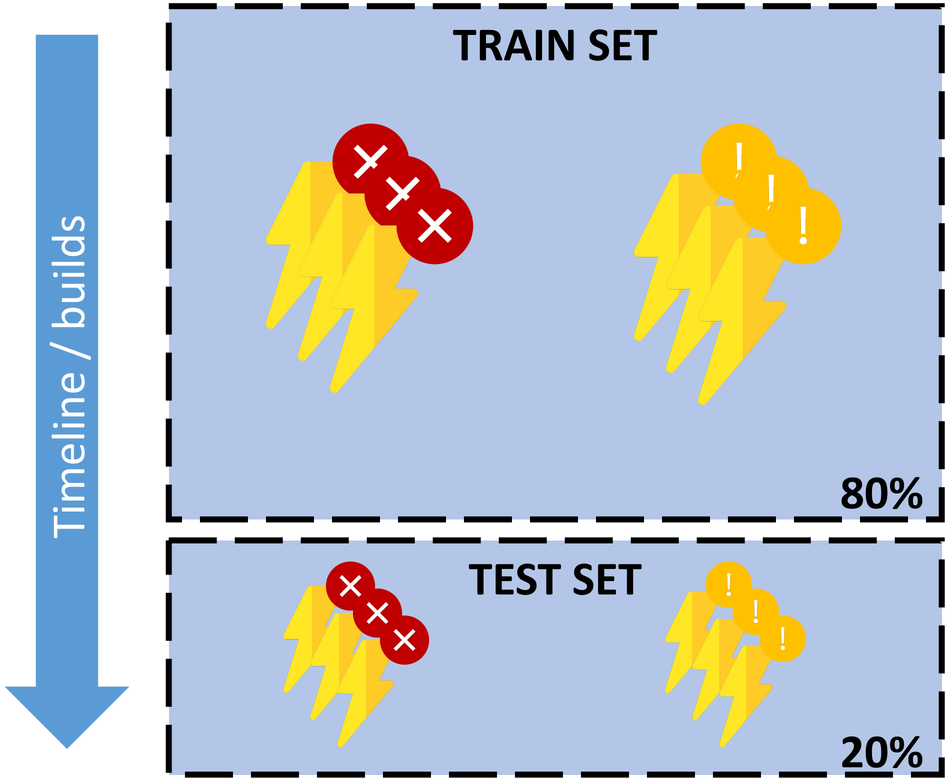

¾ of fault-triggering failures are classified as flaky (missed faults)

# Across builds




⅓ of fault-revealing tests were found to be flaky in previous builds

# 2. Focusing on failures




| Execution features | Precision | Recall | MCC | FPR |
|---|---|---|---|---|
| No | 99.7% | 91.3% | 0.25 | 20.3% |
| Yes | 99.5% | 98.7% | 0.42 | 42.3% |

**Training on failures and adding execution features improves the performance**

# Take-away messages

Approaches should focus on failures

A large part of flaky tests <u>are valuable</u> as they can <u>reveal faults</u>

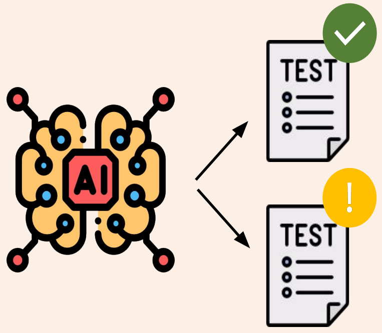
Dynamic/contextual features can help

Additional work is needed to effectively distinguish flaky from non-flaky failures

# Conclusion

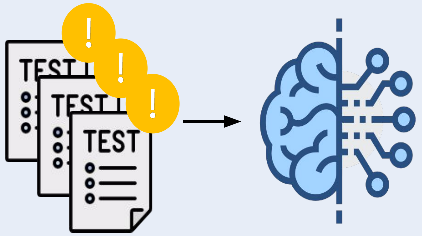
**Question I:**
Can we predict flaky tests?

- Promising
- Data needed
- Realistic validation

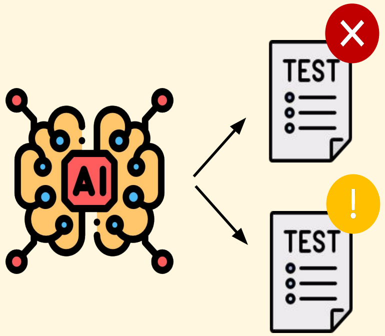



**Question II:**
Can we predict the category of a flaky test?

- Promising despite little data
- Categories support

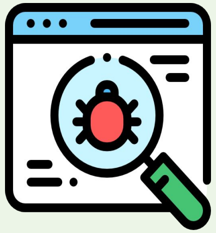



**Question IV:**
Are existing prediction techniques suitable to real-world CI?




- Difficult problem
- Focus on failures
- Consider dyn. features

**Question III:**
Can we locate the source of flakiness?



- FL for flakiness loc.
- Finer granularity?

# Challenges

**Challenge #1**
Adoption

**Challenge #2**
Comprehension

- Time-Sensitive Validation
- Intra-Project Analysis

- Chromium Case Study

- Category Prediction
- Flakiness Location

# Contributions

**Challenge #1**
Adoption

**Challenge #2**
Comprehension

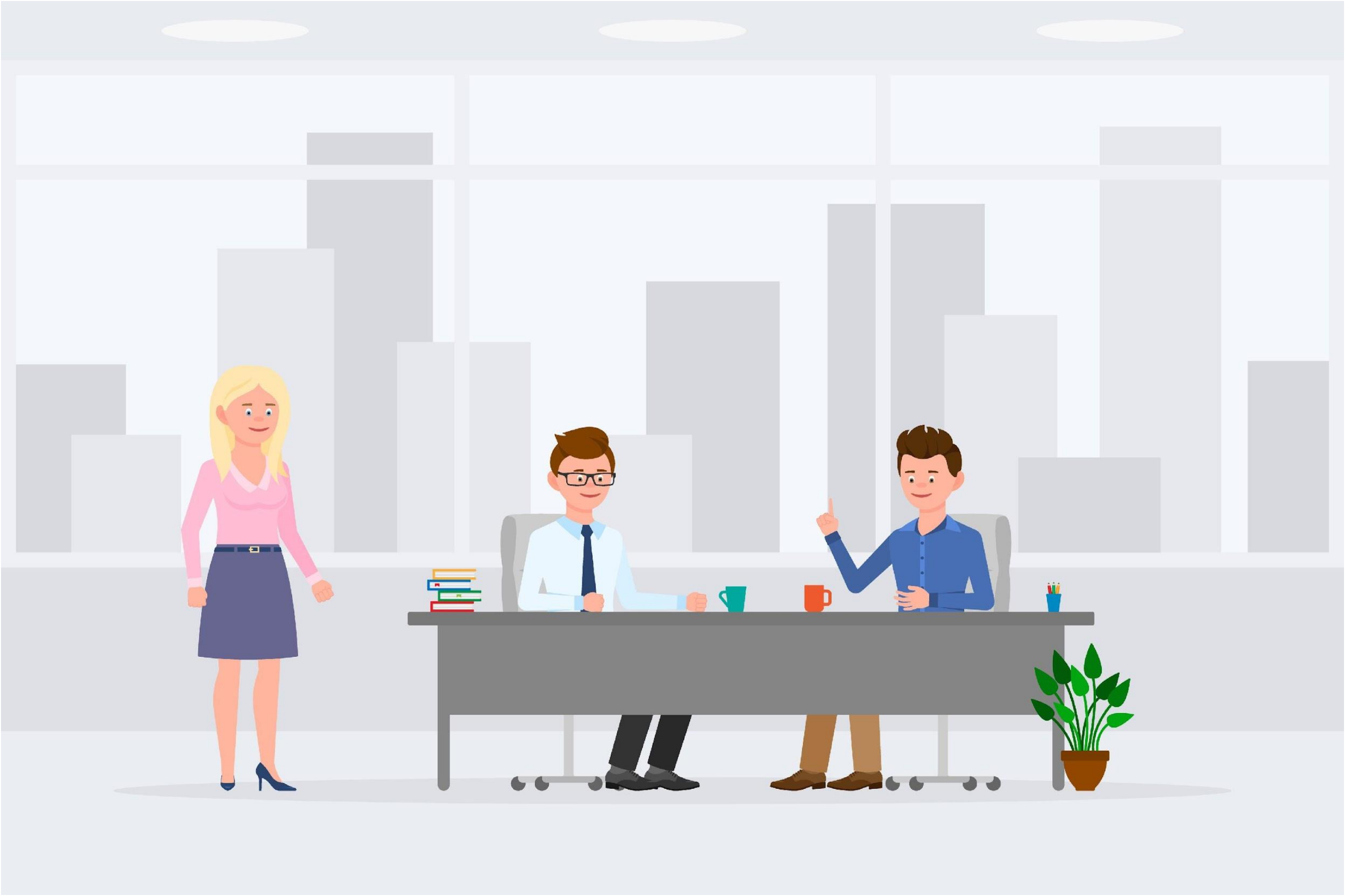
Contribution I:
A Replication Study on the Usage of Code Vocabulary to Predict Flaky Tests, **MSR** 2021

**Contribution II:**
FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning, **AST** 2023

**Contribution IV:**
The Importance of Discerning Flaky from Fault-triggering Test Failures: A Case Study
on the Chromium CI,
**under submission**

**Contribution III:**
What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness, **ICSME** 2022

# Conclusion

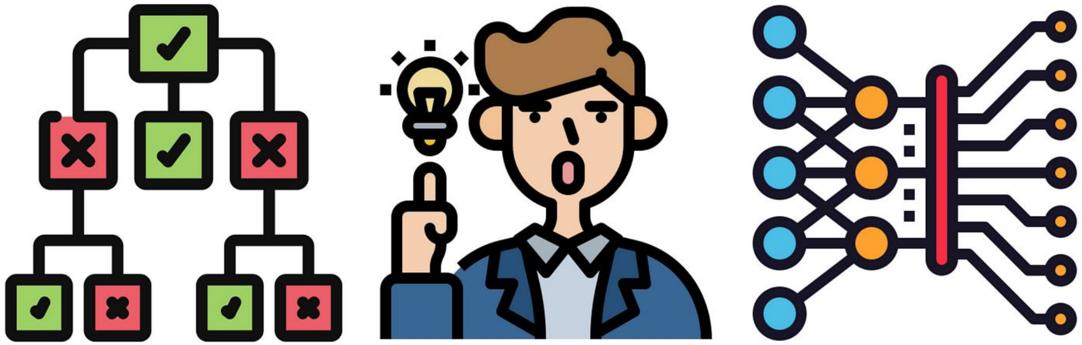


# Future work

❖ Continuity of these contributions:

- More accurate prediction and location, other features to consider

❖ Machine Learning Interpretability



❖ Practitioners studies

# List of publications

S. Habchi, G. Haben, M. Papadakis, M. Cordy, Y. Le Traon, *A qualitative study on the sources, impacts, and mitigation strategies of flaky tests*, ICST 2022

G. Haben, S. Habchi, M. Papadakis, M. Cordy, Y. Le Traon, *A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests*, MSR 2021

A. Akli, G. Haben, S. Habchi, *Predicting flaky tests categories using few-shot learning*, AST 2023

S. Habchi, G. Haben, J. Sohn, A. Franci, M. Cordy, M. Papadakis, Y. Le Traon, *What made this test flake? Pinpointing classes responsible for test flakiness*, ICSME 2022

G. Haben, S. Habchi, M. Papadakis, M. Cordy, Y. Le Traon, *The Importance of Discerning Flaky from Fault-triggering Test Failures: A Case Study on the Chromium CI*, ArXiv

Publishe
d

Under
submission

# Enabling Open Science

## Replication packages

- https://figshare.com/s/5b252c442fc36e8823cb
- https://github.com/serval-uni-lu/FlakyVocabularyReplication
- https://github.com/serval-uni-lu/FlakyCat
- https://github.com/serval-uni-lu/sherlock.replication
- https://github.com/serval-uni-lu/DiscerningFlakyFailures

## Datasets

FlakyCat, 451 flaky tests + categories
https://github.com/serval-uni-lu/FlakyCat/tree/main/data

Chromium, builds + failures information
https://figshare.com/articles/dataset/dataset/22354141

# Test Flakiness Prediction Techniques for Evolving Software Systems


Presented by:

Guillaume HABEN, University of Luxembourg, Luxembourg


Defense committee:

| | | |
|---|---|---|
| Prof. Dr. Mike Papadakis, | Chairman, | University of Luxembourg, Luxembourg |
| Dr. Maxime Cordy, | Vice-chairman, | University of Luxembourg, Luxembourg |
| Prof. Dr. Yves Le Traon, | Supervisor, | University of Luxembourg, Luxembourg |
| Prof. Dr. Arie Van Deursen, | Member & Reviewer, | TU Delft, The Netherlands |
| Prof. Dr. Javier Tuya, | Member & Reviewer, | Universidad de Oviedo, Spain |

Date:

June 29th, 2023

PayPal

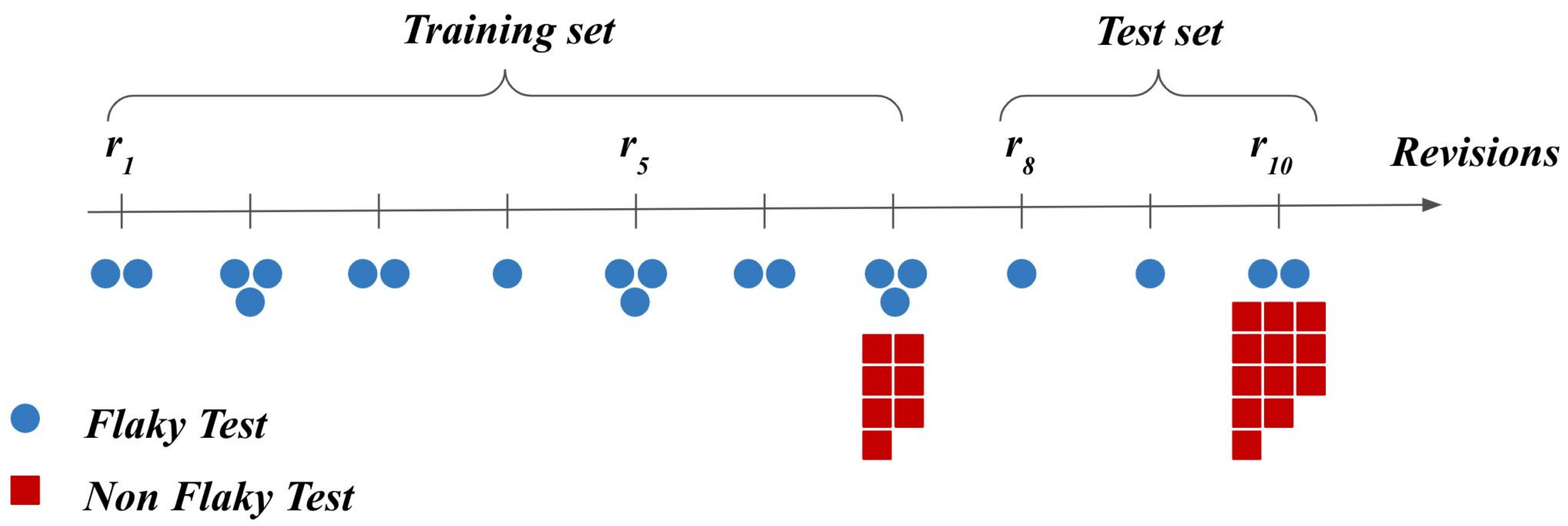
Serval securityandtrust.lu

SnT securityandtrust.lu

uni.lu UNIVERSITÉ DU LUXEMBOURG

# 1. Per project, in time and non-time sensitive settings

## Classifier (Random Forest) performance



# 2. FlakyCat performance per category

## Example of a test misclassified as Async wait

```
@Test
public void shouldPickANewServer[...]() throws
     Throwable {
[...]
    Thread thread = new Thread( () -> {
    try {
    startTheLeaderSwitching.await();
    CoreClusterMember theLeader = cluster.
        awaitLeader();
    switchLeader( theLeader );
    } catch ( TimeoutException |
     InterruptedException e ) {
        // ignore
    }});
[...]
}
```

**Commit message:**

*“A latch was being release before ensuring that the condition it was guarding for was fulfilled. This created a race that most of the time was won by the desired thread, but it was flaky.”*

# 2. FlakyCat performance per category

## Interpretation

## Example of a test misclassified as Async wait

```
@Test
public void shouldPickANewServer[...]() throws
    Throwable {
[...]
    Thread thread = new Thread( () -> {
    try {
    startTheLeaderSwitching.await();
    CoreClusterMember theLeader = cluster.
        awaitLeader();
    switchLeader( theLeader );
    } catch ( TimeoutException |
    InterruptedException e ) {
        // ignore
    }});
[...]
}
```

**Commit message:**

*"A latch was being release before ensuring that the condition it was guarding for was fulfilled. This created a race that most of the time was won by the desired thread, but it was flaky."*

Performance varies depending on the categories

# 2. FlakyCat Metrics

Then 3 possibilities:

**Micro-averaged**: all samples equally contribute to the final averaged metric

**Macro-averaged**: all classes equally contribute to the final averaged metric

**Weighted-averaged**: each classes's contribution to the average is weighted by its size (1vsAll)

# 2. FlakyCat Validation

10-fold Cross validation on the training set (75% of data)

Check performance on hold-out set (25% of data)

Unseen data in hold-out set (no leakage of oversampled elements

# 2. FlakyCat

## Test smells prediction

Journals & Magazines > IEEE Transactions on Software... > Volume: 49 Issue: 4

### Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests

**Publisher: IEEE** Cite This PDF

Sakina Fatima ; Taher A. Ghaleb ; Lionel Briand **All Authors**

Conferences > 2021 IEEE/ACM 43rd Internatio...

### FlakeFlagger: Predicting Flakiness Without Rerunning Tests

**Publisher: IEEE** Cite This PDF

Abdulrahman Alshammari ; Christopher Morris ; Michael Hilton ; Jonathan Bell **All Authors**

RESEARCH-ARTICLE

### On the use of test smells for prediction of flaky tests

**Authors:** Bruno Camara, Marco Silva, Andre Endo, Silvia Vergilio Authors Info & Claims

SAST '21: Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing • September 2021 • Pages 46–54 • https://doi.org/10.1145/3482909.3482916

**Published:** 12 October 2021 Publication History Check for updates

RESEARCH-ARTICLE

### Static test flakiness prediction: How Far Can We Go?

**Authors:** Valeria Pontillo, Fabio Palomba, Filomena Ferrucci Authors Info & Claims

Empirical Software Engineering, Volume 27, Issue 7 • Dec 2022 • https://doi.org/10.1007/s10664-022-10227-1

**Published:** 01 December 2022 Publication History

RESEARCH-ARTICLE

### Toward static test flakiness prediction: a feasibility study

**Authors:** Valeria Pontillo, Fabio Palomba, Filomena Ferrucci Authors Info & Claims

MaLTESQuE 2021: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution • August 2021 • Pages 19–24 • https://doi.org/10.1145/3472674.3473981

**Published:** 23 August 2021 Publication History Check for updates

# Genetic Programming

## Global formula

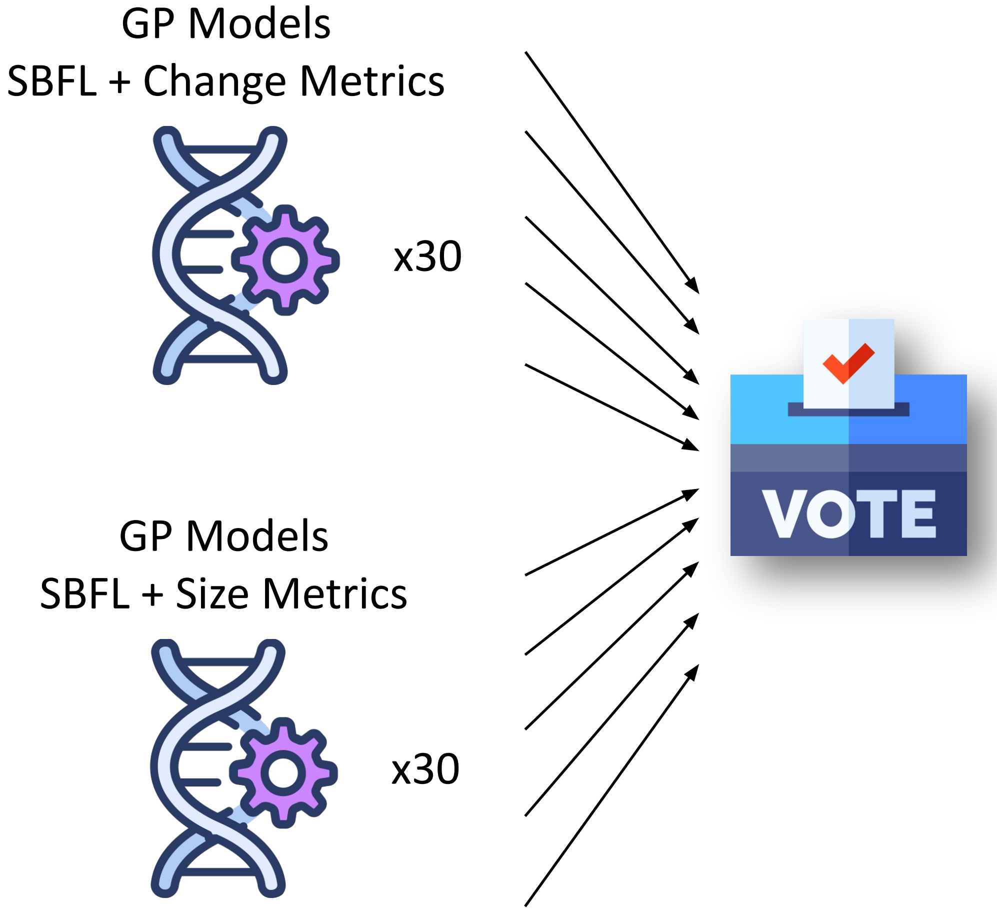



TABLE VIII: RQ3: The effectiveness of the voting between 60 different GP-evolved models, 30 from SBFL with change metrics, and 30 from using SBFL with size metrics. ‘Perc’ denotes Percentage

| **Project** | **Total** | **acc** | | | | **wef ($\mathbf{R}_{wef}$)** | |
|---|---|---|---|---|---|---|---|
| | | @1 | @3 | @5 | @10 | mean | med |
| Hbase | 8 | 3 | 5 | 6 | 6 | 9.62 (12) | 1.5 (2) |
| Ignite | 14 | 2 | 4 | 4 | 4 | 228.61 (24) | 17.5 (4) |
| Pulsar | 10 | 3 | 6 | 7 | 9 | 7.30 (12) | 2.0 (5) |
| Alluxio | 3 | 1 | 1 | 1 | 2 | 61.83 (22) | 9.0 (10) |
| Neo4j | 3 | 1 | 2 | 2 | 2 | 19.67 (42) | 1.0 (18) |
| Total | 38 | 10 | 18 | 20 | 23 | 94.61 (19) | 3.5 (5) |
| Perc (%) | 100 | 26 | **47** | 53 | 61 | - | - |

~50% flaky classes identified in the top 3

# Chromium

Table 8.5: Number of builds containing each studied test type. All builds contain flaky tests. 1/4 contain fault-revealing tests. Among the failing builds, 3/4 contain only fault-revealing tests that are flaky in other builds.

| **Builds containing** | **Number** |
|---|---|
| Flaky tests | 10,000 |
| Fault-revealing tests | 2,415 |
| Fault-revealing flaky tests | 1,974 |
| Exclusively fault-revealing flaky tests | 1,766 |

# Chromium

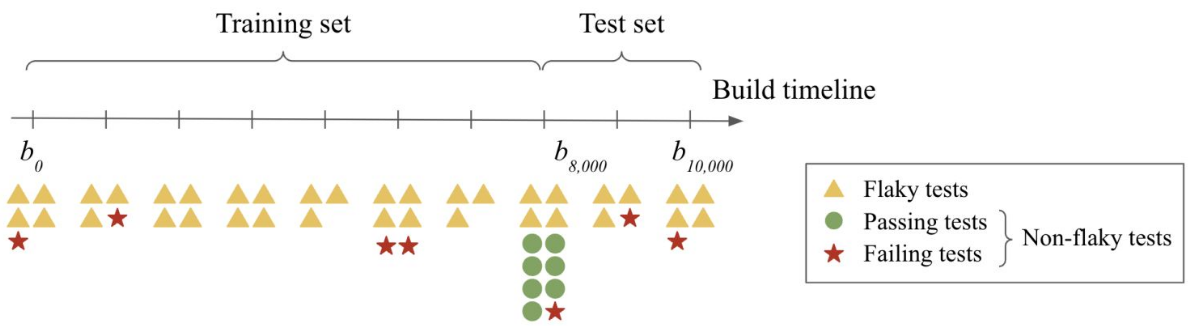
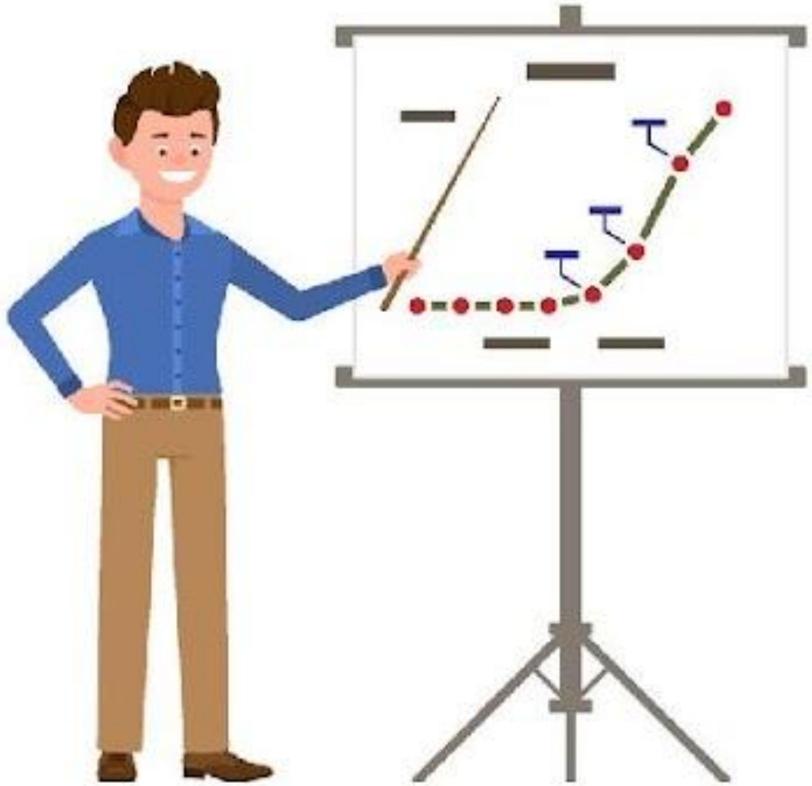
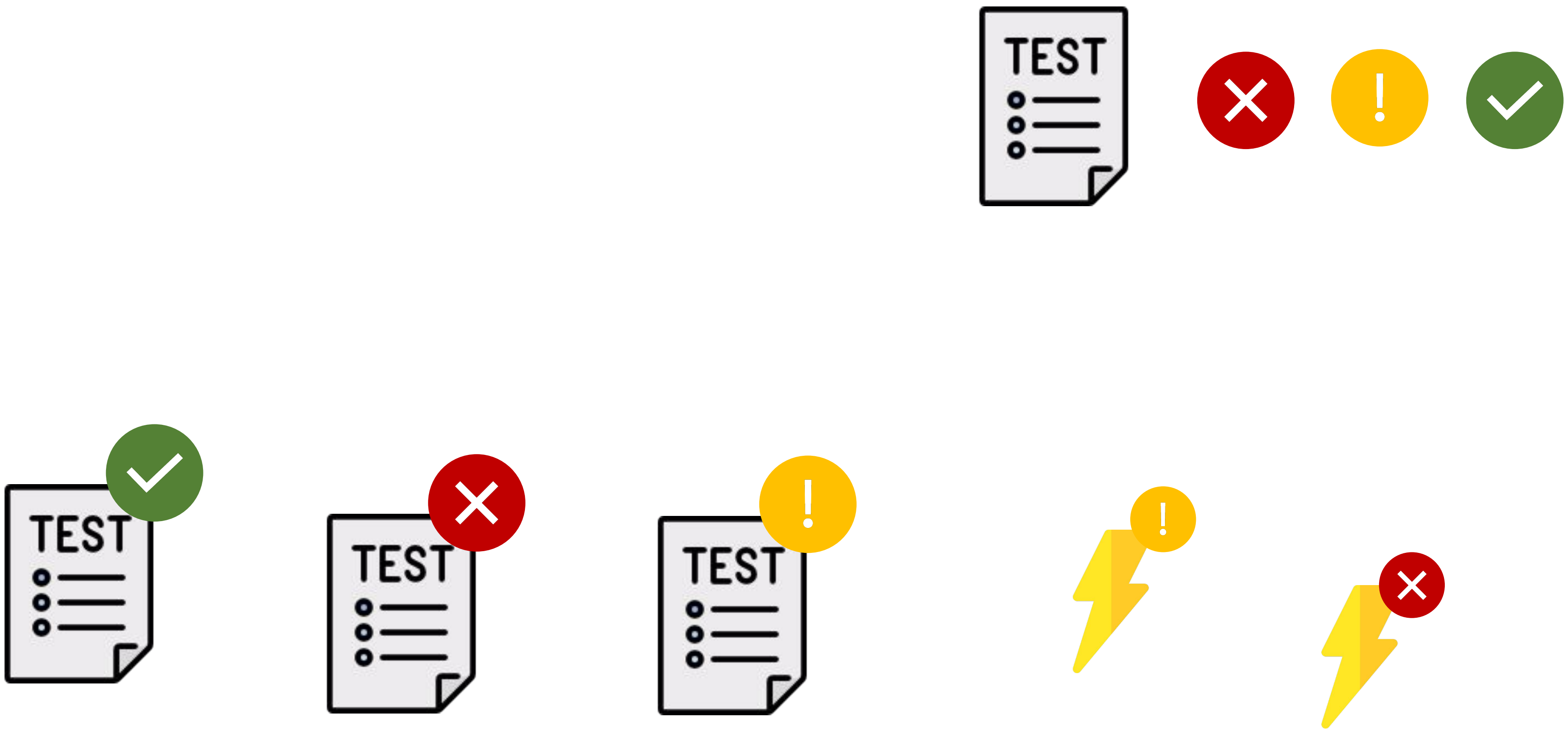



Figure 8.4: The data collected from Chromium's CI consists of flaky, fault-revealing and passing tests spread across 10,000 builds. The build timeline ranges from build $b_0$ to $b_{10,000}$ and depicts the distribution of the collected tests: flaky tests are spread across all builds and fault-revealing tests happen occasionally. Due to a large number of passing tests, we collected them from the $b_{8,000}$ build (*i.e.* at the end of our training set).

# Conclusion




TEST
TEST
TEST
TEST