An Empirical Study on the Use of Snapshot Testing

Shun Fujita
Kyoto University
Kyoto, Japan
fujita.shun.88e@st.kyoto-u.ac.jp

Yutaro Kashiwa
NAIST
Nara, Japan
yutaro.kashiwa@is.naist.jp

Bin Lin
Radboud University
Nijimegen, The Netherlands
bin.lin@ru.nl

Hajimu Iida
NAIST
Nara, Japan
iida@itc.naist.jp

***Abstract*—Testing is one of the most critical processes in software quality assurance. Developers spend a large portion of their time writing test code to avoid potential software failures. In recent years, snapshot testing, which compares snapshots of UI components to detect unexpected changes, has gained popularity in front-end development due to the need to reduce testing efforts. However, it is still unclear how software developers adopt snapshot testing and maintain them. To facilitate future work to reveal the potentials of snapshot testing, this paper presents a preliminary study which examines how developers use snapshot tests. More specifically, this study investigates 1) the characteristics of projects adopting snapshot testing, and 2) when snapshot tests were introduced and how they evolve. Our study is among the first to understand snapshot testing, providing valuable insights on its adoption. We also highlight the future directions to work on.**

***Index Terms*—Snapshot testing, JEST, Test-code, Unit tests, Empirical study**

I. Introduction

The question of "how to write good tests" has been widely studied by researchers and practitioners. Over the last few decades, lots of tools and strategies have been proposed to guide developers to test software in a systematic way at different testing levels. More recently, snapshot testing has gained popularity in front-end development due to the need of reducing testing effort. Specifically, major companies including Amazon, Google, and Microsoft have adopted snapshot testing [1].

Snapshot testing is a type of output comparison testing technique that asserts whether the outputs by the current state of the product remain unchanged. That is, it only detects differences before and after code changes, and ignores whether the current state is correct. The aim of snapshot tests differs from those of unit and functional tests that define the correct behavior of the products.

Snapshot tests require little time to create because developers can just take snapshots of UI components when the current version of the product is considered ideal [2]. The ideal state will be compared with the future state of UI components. Given the ability of detecting unexpected changes in UI, it can be applied in different scenarios. For example, React components are often shared by multiple UIs, applying snapshot testing can help ensure no dependents will break.

Currently, many snapshot testing frameworks have been created for different programming languages, such as `swift-snapshot-testing`[1] for Swift and `snapshooter`[2] for .NET. Among these frameworks, `JEST` for JavaScript and TypeScript stands out as JavaScript is one of the most popular web development languages, and JEST is highly compatible with many popular frameworks including React, Angular, and Vue.js.

While snapshot testing is widely adopted in practice and the community is still growing rapidly, researchers have paid little attention to it. It is thus still unclear how software developers adopt snapshot tests and evolve them. In this study, our goal is to understand:

> *"How do developers make use of snapshot testing along with other types of testing?"*

This study presents an empirical investigation on the use of snapshot tests, specifically JEST - the most popular testing framework for JavaScript and TypeScript. We examine the characteristics of projects using snapshot tests, as well as the adoption and evolution of snapshot tests.

Our contributions are two-fold. First, we present a new study to shed light on the characteristics and adoption of snapshot testing in open-source projects. Second, we provide a new dataset of projects adopting JEST, which can be used for future studies by other researchers. We also highlight the directions worth investigating in the future.

Replication package. To facilitate replication and further studies, the data used in this study are publicly available on GitHub repository.[3]

II. Background and Related Work

A. Snapshot Testing with JEST

A typical snapshot test case takes a snapshot of UI components and compares it with a pre-stored snapshot. The test will fail if the two snapshots do not match, meaning that there are unexpected changes. With JEST, instead of rendering graphical UI components, it generates serializable values for UI components and compares the values rather than images. The serializable values can be JSON, objects, or DOM elements (*e.g.,* React components). In the context of JEST, the term "snapshot" refers to the serializable values.

[1]https://swisslife-oss.github.io/snapshooter/
[2]https://github.com/pointfreeco/swift-snapshot-testing
[3]https://github.com/shun-fujita-hub/AnEmpiricalStudyontheUseofSnapshotTesting

JEST provides two snapshot testing methods (described below) that use different formats to record the DOM elements. For both methods, when a test fails, developers can easily re-generate the snapshot after the fix, without the need of modifying the test code thanks to the interactive mode of JEST. The details are described as follows.

`toMatchSnapshot()`: Figure 1 shows an example snippet using this method, and Figure 2 depicts an example of snapshot files generated. `toMatchSnapshot()` stores a snapshot under the "`__SNAPSHOT__`" directory. It then verifies whether the generated snapshots are the same as the ones created previously. If the method is invoked for the first time or the snapshots are deleted, snapshots are only stored and no verification will be conducted.

`toMatchInlineSnapshot()`: This method does not store snapshots in files but instead passes the serializable values of the snapshots as an argument, *e.g., toMatchInlineSnapshot(" ... ")*. The argument in the method will be automatically modified by JEST when developers update snapshots following JEST's prompt.

In this study, we refer snapshot testing to the two methods `toMatchSnapshot` and `toMatchInlineSnapshot`, as they are the only two methods used for snapshot testing in JEST. The other test assertions are treated as unit tests. Also, we do not distinguish between unit tests and integration tests.

B. Related Work

1) Testing Practice and Evolution: Aniche *et al.* [3] observed 13 developers to understand how they write tests for real-world open-source methods and surveyed 72 developers on their testing practices. The results led to a general framework and a set of strategies which explain how developers reason about testing.

Spadini *et al.* [4] analyzed over 2,000 mock usages to identify practices, rationales, and challenges of mocking objects for testing Java projects. The results show that mocks often exist since the creation of test classes and tend to remain there for the whole lifetime of test classes. Besides, changes in production code frequently drive the test code to co-evolve.

Tufano *et al.* [5] studied when test smells happen in source code and how long they are present. Their results show that test smells are usually introduced when the test code is created, and they tend to stay in a system for a long time.

2) Tests and Bugs in JavaScript Projects: Fard *et al.* [6] conducted an empirical study on 373 JavaScript projects to examine their defusion, quality metrics, and limitations. They found that 40% of client-side projects do not have a test and even if they do, the quality is moderate to low. Ocariza *et al.* [7] studied 317 bug reports of client-side JavaScript projects and found that most (65%) of bug reports are DOM-related.

Mirshokraie *et al.* [8] proposed an approach that automatically generates test oracles through a mutation-based algorithm. Their evaluation demonstrates that the approach can find injected JavaScript and DOM faults with 100% precision, and 70% recall. Their later work [9] proposed a method to generate

```
it('should position slider controller when performed drag operation on it', () => {
    const { container } = renderMinimap();
    const sliderController = container.querySelector('#sliderController') as HTMLDivElement;
    fireEvent.mouseDown(sliderController);
    fireEvent.mouseMove(sliderController);
    expect(sliderController).toHaveStyle('left: 0px');
    expect(container).toMatchSnapshot();
});
```

Fig. 1: Example of snapshot tests using `toMatchSnapshot()`

```
763   exports[`Minimap component should position slider
              controller when performed drag operation on it 1`] = `
764   .c0 {
765     position: -webkit-sticky;
766     position: sticky;
767     display: inline;
768     z-index: 5;
769     left: calc(100% - 19rem);
      ⋮       ⋮
836   .c6:active {
837     border: 2px solid #3872D2;
838   }
839
840   <div>
841     <tr
842       class="c0"
843     >
      ⋮       ⋮
1006          <div
1007            class="c6"
1008            id="sliderController"
1009          />
1010        </div>
1011      </td>
1012    </tr>
1013  </div>
1014  `;
```

Fig. 2: Example of snapshot file made by the test in Figure 1
This snapshot starts from line 763 and ends at line 1014.

unit-test assertions using existing GUI tests and the method can find 26% more faults than their previous method [8].

Christophe *et al.* [10] analyzed test code in projects using Selenium (a web application testing tool) and found that it takes 11.23 non-Selenium commits (or 4.33 days) on average before a commit affects a Selenium file.

3) Snapshot tests: While various types of tests including unit tests, integration tests, and GUI tests have been studied by researchers [3] [4] [5] [10], only a few of them [2] [11] have focused on snapshot testing.

Bui and Rocha [11] published a dataset of repositories using JEST's snapshot files. They identified JEST adoption by checking whether the GitHub repository uses the tag of "jest". Their data includes attributes from the latest commit of projects (*e.g.,* the number of stars, issues, snap files, test files).

Cruz *et al.* [2] conducted a grey literature review in order to investigate the adoption of snapshot testing in practice, revealing trade-offs and best practices. They found that snapshots are simple to create and can prevent regressions while they may easily become fragile if used improperly.

III. Study Design

A. Research Questions

Our study aims to answer the following research questions:

RQ_1: What are the characteristics of projects that adopt snapshot tests? As JEST's official document[4] states, *"the aim of snapshot testing is not to replace existing unit tests, but to provide additional value and make testing painless."* We conjecture that projects adopting snapshot testing might have different testing behaviors. Therefore, we propose this RQ to understand whether different characteristics can be disclosed between the projects adopting and not adopting JEST for snapshot testing.

RQ_2: When are snapshot tests introduced and how do they evolve? Testing is notorious for being a time-consuming software development process and is tedious work for developers [12]. Snapshot tests are developed to mitigate this challenge and are widely used by practitioners. However, it is still unclear when these snapshot tests are created and how they are maintained. In this RQ, we aim to understand how snapshot tests evolve.

B. Data Collection

In our study, we mainly target the projects adopting JEST, an all-in-one testing framework. JEST is reported to be one of the most popular testing frameworks in the web development.[5] The following subsections introduce steps to collect snapshot tests in JEST.

1) Identifying projects using JEST: As described in Section II-B, a previous study [11] provides a list of JEST projects. However, we decided to collect repositories that use JEST ourselves because they only collected repositories with the tag "jest" and we expect that most projects would not declare what test framework they employ with GitHub tags.

Instead of using the "jest" tag, we use the following approach to identify the projects using JEST. We first obtained the list of non-forked JavaScript or/and TypeScript repositories with more than 1,000 stars using GitHubSearch [13], which returned 9,516 repositories. We then cloned all the repositories and parsed the `package.json` file to determine whether JEST is used. Specifically, we check if there is at least one "jest" command in the "script" section.

2) Identifying snapshot tests and unit tests: After collecting the repositories using JEST, we need to identify which test methods use snapshot testing. We used Babel parser,[6] a popular AST parser for JavaScript and TypeScript, to get the AST node of each test method. We used AST parser as it could easily filter out commented-out methods. Once the test method node (*i.e.,* the method whose name starts with `test` or `it`) is identified, we further examine if `toMatchSnapshot` or `toMatchInlineSnapshot` is used in the test method, as they are the only methods provided by JEST for snapshot testing. Test methods containing at least one snapshot test assertion (*i.e.,* `toMatchSnapshot` or `toMatchInlineSnapshot`) are considered as *snapshot-tests*. The remaining test methods containing only non-snapshot assertions, such as `toBe` and `toEqual`, are considered as *unit-tests*.

[4]https://jestjs.io/docs/snapshot-testing

[5]https://2022.stateofjs.com/en-US/libraries/testing/

[6]https://babeljs.io/docs/babel-parser

TABLE I: Dataset summary

repositories	#	%
JEST repositories	1,487	-
using snapshot tests	569	38.2%
only file snapshots	370	65.0%
only inline snapshots	93	16.3%
both of them	106	18.6%

Dataset Summary: Table I shows the number of repositories that use JEST framework. We identified 1,487 repositories using JEST and found that 15.6% of JavaScript or/and TypeScript projects use JEST test methods (= 1,487/9,516) and 38.3% (= 569/1487) of JEST projects use snapshot tests. Also, a significant number of projects (18.6%) use *inlined-snapshots* in addition to *file-snapshots*.

Our dataset contains 569 repositories that have over 1000 stars and use snapshot testing. If we apply the same filtering strategy ($>$ 1000 stars), the previous dataset [11] would have only nine repositories left. Besides, they identify snapshots by locating ".snap" files. This approach ignores *inlined-snapshots*, which account for 16.3% of snapshot tests according to our analysis. We believe that our dataset provides a list of projects which adopt snapshot testing and have a higher quality, which can be used in future snapshot testing related studies.

C. Analysis

1) RQ_1 (characteristics of projects): To answer RQ_1, we grouped the collected repositories into three categories: repositories using only *unit-tests* (`UT`), repositories using only *snapshot-tests* (`ST`), and repositories using both *unit-tests* and *snapshot-tests* (`UT+ST`). We measured the following metrics for each project and then compared their median values.

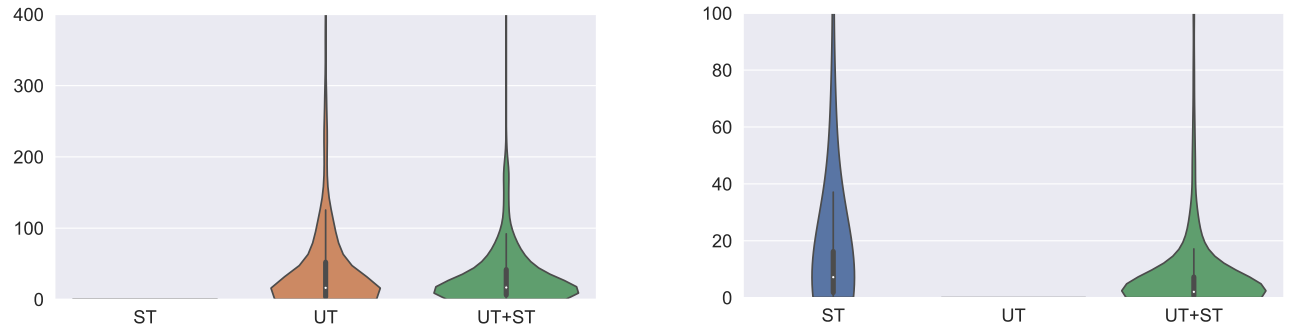
- *Normalized Number of Test Cases:* The number of test methods per 1,000 lines of production code in each project. We consider those coding files, which do not contain "test" in the file names or the names of their directories, as production code. Kochhar *et al.* [14] suggested that the number of test cases is weakly correlated with the number of bugs. Therefore, it would be interesting to inspect the test cases these projects have.
- *Number of Assertions per Test:* The average number of assertions per test method. This metric is inspired by the study indicating that a larger number of assertions is correlated with a significantly smaller number of faults [15].

We apply the Mann-Whitney U-test to examine whether there is a statistically significant difference when using different pairs of project groups (*e.g.,* `UT` vs. `ST`). Note that we apply the Bonferroni correction to the tests to prevent increases in the family-wise error rate (*i.e.,* α = 0.005).

(a) Number of test cases

(b) Number of test assertions

Fig. 3: Distribution of metrics for each type of project



(a) Number of unit test cases

(b) Number of snapshot test cases

Fig. 4: Distribution of the number of unit and snapshot tests

2) RQ_2 (snapshot test introduction and evolution): We investigate when *snapshot-tests* were introduced and how they evolved. Like the previous study by Spadini *et al.* [4], we examined when developers introduced snapshot tests after they created the test code. More specifically, for each test method in the latest commit, we first identified which commit introduced them. We then further inspected if the method was created simultaneously with the file creation. Finally, we classified them into two types: *snapshot-tests* introduced from the file creation and *snapshot-tests* introduced later. As for their evolution, we examined the percentage of commits updating snapshot files out of all the commits after the first snapshot file was pushed. Also, we studied what kind of files are likely to be updated with snapshot files (*i.e.,* co-changes).

IV. Results

A. RQ_1: What are the characteristics of projects that adopt snapshot tests?

Figure 3 depicts the distribution of the numbers of test cases/1k LOCs and numbers of assertions/test method for the studied projects. After categorizing projects based on the type of tests, we found 1,029 repositories with only unit tests (`UT`), 567 repositories with both unit and snapshot tests (`UT+ST`), and only 31 repositories with only snapshot tests (`ST`). Their median sizes of production code (LOCs) are 3,622, 1,371, and 5,404 for `UT`, `ST`, and `UT+ST`, respectively. This implies that larger projects tend to use both snapshot and unit tests.

Regarding the number of test cases (normalized by 1K-LOCs), `UT+ST` has a statistically significantly larger median number of test cases than the others: 25.2 (`UT+ST`) vs. 16.2 (`UT`) / 7.2 (`ST`). However, when we look into the number of unit tests and snapshot tests for each type of projects (Figure 4a), we observed that the median numbers of unit tests are almost the same (16.2 in `UT`, 16.7 in `UT+ST`), which implies developers use snapshot tests in addition to unit tests. Our Mann-Whitney U-test shows a no statistically significant difference can be found between `UT` and `UT+ST`, with a p-value of 0.23.

When comparing the number of snapshot tests in `ST` and `UT+ST` (Figure 4b), `ST` has 3.4 times more snapshot tests (`ST`: 7.2, `UT+ST`: 2.1) even though the total number of test cases is smaller (shown in Figure 3a). The projects using snapshot tests might mitigate testing efforts by using snapshot tests.

As for the number of assertions in Figure 3, projects in `UT` and `UT+ST` groups have a similar number of assertions (UT: 1.62, UT+ST: 1.7). Also, we observed that repositories in `ST` have a median of only one assertion in their test methods.

RQ_1. Projects using both unit tests and snapshot tests have 2.6 times more test cases than projects using only unit tests while the numbers of unit tests in both groups are almost the same.

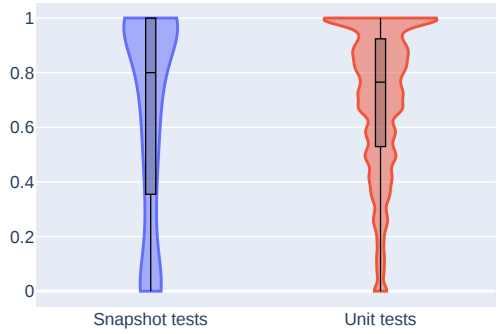


Fig. 5: Percentage of snapshot files created from the beginning
(Note: 2% of repositories cannot be calculated due to their size)

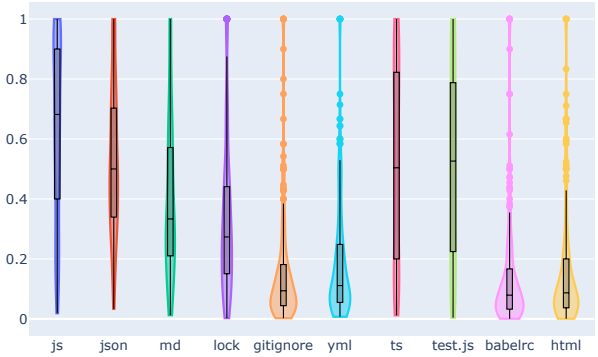


Fig. 6: Percentage of co-changed files with snapshot files

B. RQ_2: When are snapshot tests introduced and how do they evolve?

Figure 5 shows the ratio of snapshot tests and unit tests that are created at the same time as file creation. We discovered that about a median of 80.0% of snapshot tests and 76.5% of unit tests are created at the same time as file creation. The percentage for snapshot tests is slightly higher than that of unit tests. It might be because developers may utilize snapshot tests as a quick/instant testing approach and then add unit tests to complement snapshot tests and cover the missing part. For example, we found that unit tests were added on top of snapshot tests to reach a 100% coverage.[7]

As for the evolution, we found that a non-negligible number of commits (8.2%) update the snapshot files. Figure 6 shows what percentage of files of different types are co-changed with snapshot files in each project. Due to space limitations, we only show the top 10 file types that are commonly modified by studied projects. From the figure, we can see that snapshot files are often updated along with js, ts, and test.js (test) files. Also, JSON and MD files are also frequently modified together. In future work, we plan to further investigate what kind of co-changes are performed instead of only looking into file types.

RQ_2. A non-negligible number of commits (8.2%) update the snapshot files. Co-changes with snapshots are common, especially with js, ts, and test.js (test) files.

V. Threats to Validity

Internal validity: Our study uses `packages.json` to identify repositories using JEST. This file is generated by popular package managers like NPM and YARN, and other tools might not produce this file. Thus, we might miss some repositories using JEST. However, we do not expect the number to be huge and it equally impacts the different categories in our analysis.

Construct validity: While checking when snapshot tests were introduced, we did not consider the case of refactoring. Being unable to detect refactorings like Rename Method might lead to mistakes in the creation date. However, the issue would also exist equally in different groups of our analysis. We plan to integrate refactoring detection tools in the future.

External validity: Our study mainly focuses on JEST. Indeed, there are several frameworks which support snapshot testing in different programming languages. It is unclear whether our observation still holds for other frameworks and further studies are needed.

VI. Future Directions

Our study revealed that projects often use snapshot tests along with other types of tests. While snapshot testing has been widely used, it is still unclear to what extent snapshot testing can benefit developers. In the future, we plan to look into the following directions.

Efficiency of snapshot tests. In RQ_2, we observed that developers add unit tests to test files containing snapshot tests to increase coverage. Mockus *et al.* [16] observed that an increase in test coverage correlates with a decrease in field-reported problems. Thus, we plan to investigate (i) whether repositories using both snapshot tests and unit tests have higher coverage than those using only unit tests, (ii) what percent of code is covered by both unit tests and snapshot tests, and (iii) which tests can help to increase coverage more quickly. However, to investigate these questions, a lot of effort is needed to run tests and coverage tools as many projects have their own specific testing environments and steps. For example, a studied project[8] requires preparing specific software installations and running a pre-test command before testing.

Effectiveness of snapshot tests. Depending on the type of test (*e.g.,* combination tests and unit tests), the variety of detected bugs is different. However, the effective cases for snapshots are not empirically studied. Hence, we plan to perform manual inspections of bug reports in order to investigate what bugs can/cannot be detected by snapshots compared with other tests. Our ultimate goal is to recommend appropriate test methods to developers, based on the context.

Optimal practice of snapshot testing. We would like to survey developers to understand of issues they encounter with snapshot testing and the good practices applied.

[7]https://github.com/ctrlplusb/react-universally/commit/3623eae1b18704b90e499a8fbbd5f1dd3e82da1e

[8]https://github.com/juliomrqz/statusfy/blob/master/package.json

VII. Conclusion

We conducted an empirical study on 1,487 projects using JEST (out of which 569 adopt snapshot testing) to understand how snapshot testing is used. We observed that (i) projects using both unit tests and snapshot tests have much more test cases than projects using only unit tests although they have similar numbers of unit tests; (ii) A non-negligible number of commits (8.2%) update the snapshot files, and co-changes with snapshots occur frequently. Our study shed light on how snapshot testing is adopted in open-source projects, and the provided dataset can facilitate relevant future studies.

Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP21K17725) and JST for the PRESTO grant (JPMJPR22P3).

References

[1] Meta. https://engineering.fb.com/2022/05/11/open-source/jest-openjs-foundation/.

[2] V. P. Gazzinelli Cruz, H. Rocha, and M. T. Valente, "Snapshot testing in practice: Benefits and drawbacks," *Journal of Systems and Software*, vol. 204, p. 111797, 2023.

[3] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2022.

[4] D. Spadini, M. F. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems - why and how developers use them, and how they evolve," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1461–1498, 2019.

[5] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 4–15.

[6] A. M. Fard and A. Mesbah, "Javascript: The (un)covered parts," in *Proceedings of the 2017 International Conference on Software Testing, Verification and Validation (ICST'17)*, 2017, pp. 230–240.

[7] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *Proceedings of the 2013 International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, 2013, pp. 55–64.

[8] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "JSEFT: automated javascript unit test generation," in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST'15)*, 2015, pp. 1–10.

[9] ——, "Atrina: Inferring unit oracles from GUI test cases," in *Proceedings of the 2016 International Conference on Software Testing, Verification and Validation (ICST'16)*, 2016, pp. 330–340.

[10] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 141–150.

[11] E. Bui and H. Rocha, "Snapshot testing dataset," in *Proceedings of the 20th International Conference on Mining Software Repositories (MSR'23)*, 2023, pp. 558–562.

[12] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE'14)*, 2014, pp. 19–29.

[13] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *Proceedings of the 18th International Conference on Mining Software Repositories (MSR'21)*, 2021, pp. 560–564.

[14] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *Proceedings of the 13th International Conference on Quality Software*, 2013, pp. 103–112.

[15] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 Joint Meeting on Foundations of Software Engineering (FSE'15)*, 2015, pp. 214–224.

[16] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*, 2009, pp. 291–301.