Performance Testing in Open-Source Web Projects: Adoption, Maintenance, and a Change Taxonomy

Sergio Di Meglio §, Luigi Libero Lucio Starace §, Valeria Pontillo ¶, Ruben Opdebeeck ¶,
Coen De Roover ¶, Sergio Di Martino §
§ Department of Electrical Engineering and Information Technology, University of Naples Federico II, Italy
Email: (sergio.dimeglio, luigiliberolucio.starace, sergio.dimartino)@unina.it
¶ Software Languages (SOFT) Lab, Vrije Universiteit Brussel, Belgium
Email: (valeria.pontillo, ruben.denzel.opdebeeck, coen.de.roover)@vub.be

***Abstract*—Performance testing is crucial to ensuring that web applications meet user expectations under varying workloads. Activities such as stress, load, and smoke testing are designed to simulate different kinds of simultaneous user interactions and assess system behavior. Despite its recognized importance in quality assurance of large-scale web-based systems, witnessed by numerous studies proposing solutions to support these activities, the real-world adoption and evolutionary dynamics of performance tests have received limited attention in the literature. To fill this gap, we analyzed 77 open-source web projects using APACHE JMETER and LOCUST. Our study investigates how performance tasks are performed (adoption time, load design, types of tasks), the characteristics of projects that adopt them, and their long-term maintenance. Our findings reveal that performance tests in open-source projects are simple, with a focus on single-user behaviors and minimal requests, and most tests have low concurrency. Load tests are the most common, followed by smoke and stress tests. Projects with performance tests tend to be larger and more actively maintained. However, tests are mostly long-lived but rarely updated, suggesting potential risks to their relevance and coverage over time. Finally, by creating a taxonomy of performance test changes, we observe recurring patterns of modifications, including workload adjustments, network request changes, and updates to system monitoring.**

***Index Terms*—Performance Testing, Test Maintenance, Web applications, Empirical Study.**

I. INTRODUCTION

As web applications become increasingly integral to daily activities such as banking and shopping, their ability to handle high workloads is critical. Performance issues such as slow response times, memory leaks, and lock contention, can significantly degrade the user experience, leading to customer dissatisfaction and financial or reputational losses for companies [1], [2]. In this context, performance testing can play a key role in evaluating how well web applications perform under varying workloads. In general, performance tests generate representative workloads by simulating a number of concurrent users. For web applications, this typically means sending network requests to a web server [3].

Performance testing includes various activities, such as load testing, stress testing, and smoke testing. Each activity is designed to identify specific load-related issues by running the system under different workload conditions. Despite the differences, all activities follow the same three-step process: test design, execution, and analysis. The first phase is used to design the simulated workload. In the test execution phase, tools such as APACHE JMETER and LOCUST generate the workload by replicating user interactions and collecting system metrics such as response time or CPU usage. In the last phase, the collected data is examined to identify any potential problems and ensure that performance goals are satisfied.

Despite the importance of performance testing in quality assurance of large-scale web-based systems, as witnessed by experience reports [4]–[6], the actual adoption of performance tests and their evolution over time have received limited attention in the literature. To the best of our knowledge, Leitner and Bezemer [6] are the only ones to have studied the adoption of performance tests in open-source JAVA projects. The long-term maintenance of these tests has not yet been studied.

To address this void, we present an empirical study of the adoption and maintenance practices that govern performance testing in open-source web applications. Starting from a recent dataset [7] comprising 226 performance tests from 77 open source web repositories, we investigate how performance testing is conducted in practice, such as the manner in which workloads are defined. Furthermore, we explore the characteristics of repositories that adopt performance testing, shedding light on the types of projects most likely to implement these practices. Finally, we analyze the maintenance of performance tests by studying the frequency at which they are changed and all the changes made to the tests in our dataset. This analysis enables us to create a taxonomy of common test changes.

Our results show that performance testing in open-source projects is often delayed, with adoption increasing significantly only after 2018. While some projects integrate performance tests early, the majority introduce them later, which suggests that performance concerns only come later. Regarding workload design, the tests tend to have a very short duration and simulate a low number of concurrent users, rather than generating a realistic workload. Additionally, we observe misclassifications of test types, indicating inconsistencies in test organization and a lack of standardized practices. We also find that performance tests are generally long-lived but rarely updated, with many tests persisting for years without modifications, raising concerns about their long-term relevance.

Finally, through the systematic analysis of test evolution, we identify ten main categories of performance test modifications,

with workload shape adjustments, network request updates, and assertion refinements being the most frequent.

In summary, this paper makes the following contributions:

1) An empirical analysis of performance testing adoption in 77 open-source web applications, focusing on how performance testing is conducted, the characteristics of projects that integrate these tests, and the maintenance of the performance tests;
2) A taxonomy of common performance test changes coming from the analysis of 787 individual changes, consisting of ten categories and 18 sub-categories;
3) An online appendix [8] providing all data used to conduct our study, enabling replication and extension of our research by the community.

II. Background and Related Work

Performance testing ensures that the system under test (SUT) behaves as expected under different workloads. For web applications, a workload is equivalent to multiple concurrent users making requests to a web server [3], [9]. By simulating these workloads, testers can analyze system performance and identify potential problems, such as slow response times, resource bottlenecks, or failures under high-stress conditions [10].

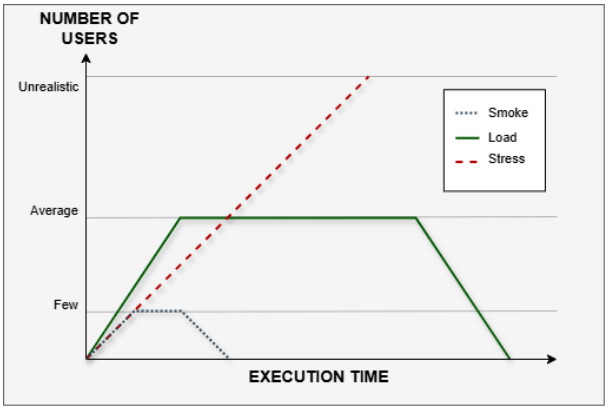

Figure 1: Overview of the different types of performance tests according to the usual form of workload defined in terms of duration and number of concurrent users.

Performance testing is commonly understood as synonymous with load, smoke, and stress tests [11]. Each type of performance test assesses a system's performance under a different type of workload, as illustrated by Figure 1:

Load tests aim to verify the behavior of the SUT under realistic load conditions. To this end, they simulate interactions between a controlled number of concurrent users that reflect typical usage patterns [5], [12].

Stress tests aim to push a system to its limits to assess its robustness and determine the maximum load it can handle [13]. Extreme conditions may include excessive load levels [14] or resource limitations, such as high CPU usage or infrastructure failures [15]. Stress tests are also used to assess the robustness of software architectures, ensuring that the system remains responsive under extreme conditions [11], [16].

Smoke tests are load tests with a minimal load, used to verify that a system is up and running and to establish performance baselines. Their workload involves but a few concurrent users for a short period of time, typically not exceeding 30 seconds [6], [17].

A. Brief Introduction to Performance Testing Phases

While the aforementioned types of performance tests serve different purposes, their design, execution, and analysis is similar [5], [18]:

Test Design: According to the literature, there are two approaches to creating a workload. The first focuses solely on making requests according to a specific target rate. For instance, simulating a certain number of immediate purchase requests in a given time interval [5], [19]. The second approach simulates user interactions more realistically through a sequence of requests that reflects typical user behavior. For example, first logging in, then some browsing, followed by the eventual purchasing —including periods of inactivity in between [5], [18].

Test Execution: A workload's execution can be automated through a load generation tool [5]. The most popular open-source tools are JMeter and Locust [20]. JMeter enables manual workload configuration through a graphical user interface [21]. Its core abstraction is the so-called `Thread Group` which represents one or more concurrent users that perform the same actions. Thread Group properties include the number of concurrent users and how often or for what duration each user repeats their actions [22]. Alternatively, Locust enables algorithmic workload configuration through Python scripts [23]. Its core abstraction is the `TaskSet` which simulates user activity as a set of tasks (requests). Additional TaskSet properties include the number of concurrent users and its duration, which can also be specified via the command line when the test is run [24].

Test Analysis: Once a test has been executed, system metrics such as CPU usage, memory usage, or throughput, are analyzed to assess whether the system satisfies its performance objectives. As manual analysis is infeasible for large volumes of data, heuristics are commonly used to automatically detect threshold violations, recognize behavioral anti-patterns, or flag anomalies by comparing observed behavior against expected norms [4], [5], [25].

B. Related Work

Weyuker et al. [26] presented a hands-on perspective on load testing, outlining its objectives and the role of requirements therein, and demonstrating how to conduct load tests through concrete examples. Hassan et al. [4] analyzed the challenges of applying performance testing in industrial environments, emphasizing the complexity of designing realistic workloads. They highlighted how changes in user behavior and functionality often render test workloads outdated, complicating

long-term test reliability. They also discussed the difficulties of determining pass/fail criteria for load tests, especially within modern DevOps practices where frequent executions are required. Avritzer et al. [27] proposed best practices for defining system-independent workloads for performance evaluation. More recently, Trani et al. [28] investigated testing activities in an agile setting, in collaboration with an industry partner. Their findings reveal that agile performance testing is still immature, with practitioners resorting to outdated and inadequate practices.

Despite the valuable insights brought by the aforementioned studies, little is known about the adoption and long-term maintenance of performance tests in open-source projects. While these aspects have been studied extensively for unit tests [29]–[32], and to some extent for GUI tests of web applications [33], the work by Leitner and Bezemer [6] is the sole that has explored the adoption of performance tests —but not their maintenance. The authors analyzed the adoption and practices of performance testing in JAVA-based open-source projects from GITHUB, focusing on non-web contexts. Their findings reveal that performance tests are scarce and typically written by core developers as isolated, one-off activities.

Our work aims to fill this void by leveraging a recent corpus of web applications with performance tests[1] to analyze the extent of their adoption, by what type of projects, in what manner —as well as the types of changes they undergo during maintenance activities.

III. GOAL AND RESEARCH QUESTIONS

The *goal* of this study is to investigate the adoption and maintenance of performance tests in open-source web applications. The *purpose* is to understand how performance testing is conducted in practice, which web applications decide to adopt this testing practice, and how performance tests are maintained over time. The *perspective* is that of both researchers and practitioners. The former are interested in understanding the current practice of and open problems in the maintenance of performance tests, whereas the latter are interested in actionable insights to improve their test and maintenance strategies.

Our study focuses on four research questions. First, we aim to investigate how performance testing is conducted in open-source web applications by examining three aspects: adoption, the types of performance tests used [5], and the design of their workloads. The latter helps understand challenges in workload modeling. So, we ask:

RQ$_1$. *How is performance testing conducted in open-source web applications?*

We continue our analysis by investigating whether the nature and characteristics of the project influence the adoption of performance testing practices. Specifically, we aim to verify possible correlations between the adoption of performance tests and metrics such as number of lines of code and the number of contributors. So we ask:

RQ$_2$. *What are the main characteristics of projects adopting performance testing?*

Then, we investigate the maintenance of performance tests by analyzing how they change over time. To do so, we examine whether performance tests are actively maintained or eventually abandoned, providing insights into the long-term sustainability of performance testing practices. So, we ask:

RQ$_3$. *How frequently are performance tests changed?*

Finally, we focus on the specific modifications performance tests undergo throughout their lifetime. By identifying recurring patterns in test modifications, we aim to offer practical insights that help practitioners anticipate maintenance challenges and improve test longevity. So, we ask:

RQ$_4$. *How are performance tests modified?*

Context of the Study: The context of our study is a dataset of 5,563 non-trivial web applications featuring end-to-end tests [7]. Since our study focuses exclusively on performance testing in web applications, we explicitly select only those with tests designed to generate workloads that simulate users sending network requests (e.g., `HTTP`, `WSS`, etc.) to a backend. We identified 66 repositories containing 201 APACHE JMETER tests and 12 repositories with 25 LOCUST tests, as shown in Table I. Notably, one of the repositories adopts both tools, providing an interesting case for further investigation.

Table I: Overview of repositories and tests collected from the dataset, classified according to frameworks and tools used for performance testing.

Performance Testing		
Load Generator Tool	**Num. of Repos**	**Num. of Tests**
APACHE JMETER	66	201
LOCUST	12	25

IV. RQ$_1$: HOW IS PERFORMANCE TESTING CONDUCTED IN OPEN-SOURCE WEB APPLICATIONS?

A. Research Method

To answer **RQ$_1$**, we analyze the 226 performance tests present in the web applications to assess the adoption of performance testing, the characteristics of the performance tests, and the performance testing activities conducted. In particular, our research method consists of three steps:

Adoption of performance testing over time. In this step, we identify when performance testing activities have been integrated into the software development lifecycle. To do so, we compare the date of the first commit in which the web application was created with the date of the commit that first introduced a performance test.

[1]Reference omitted for double-anonymous reasons.

Characteristics of performance tests. To provide a detailed characterization of the performance tests, the first two authors of the paper (a PhD student and a postdoc researcher with three years of experience in the field), qualitatively analyze the tests. To this end, they manually inspect the latest version of the tests in the APACHE JMETER GUI and a Python IDE for the 201 APACHE JMETER and 25 LOCUST tests, respectively. The investigation focuses on the following aspects:

- *Workload Design:* We analyze the number of enabled user behaviors that represent the workload. Specifically, for APACHE JMETER, we count the number of `Thread Group` or similar plugins, while for LOCUST we count the number of `TaskSet` or similar classes (cf. Section II). Then, for each user behavior, we count the number of network requests, providing insights into the workloads typically adopted in open-source projects.
- *Maximum number of concurrent users and duration:* Whenever possible, we analyze the peak of concurrent users and the duration of each test. For APACHE JMETER, we extract this information from the test code unless the values are provided externally (e.g., via command line or external files), so with no default values. In cases where dynamic variables are used (e.g., `${__P(threads,10)}`), we consider the default value. We also distinguish between the duration specified as iterations of a loop and the duration in seconds. We do not consider performance tests when it is not possible to understand and evaluate the precise duration, e.g., both iteration count and duration are set within the `Thread Group`. For LOCUST, we do not collect these values because the configuration is provided via command line.
- *System behavior metrics:* During performance testing, it is important to analyze and understand the system's health and its reaction to the test. For this reason, we extract information from key system behavior metrics collected by different plugins, which fall into three categories: (i) request status metrics, such as the number of successful or failed requests and error rates; (ii) time-related metrics, including throughput, average response time, and latency; and (iii) resource usage metrics, which capture system resource consumption, such as CPU and memory usage. It is important to emphasize that some repositories might rely on external frameworks for monitoring system behavior, which our analysis could not capture.

Classification of performance testing activities. To gain more insights into the different kinds of performance testing activities adopted, we classify tests into three categories: smoke, load, and stress, using the criteria below as general guidelines.

- In line with the typical interpretation [6], [17], [34], [35], we classify smoke tests as performance tests with no more than 2 concurrent users and a duration of at most 10 loop iterations or 30 seconds.
- Stress tests are classified if they meet at least one of the following criteria: (1) the test name or directory path includes the term *"stress"*, (2) the test runs for exceptionally long or infinite durations, or (3) the test executes until predefined shutdown constraints are met.
- Load tests are identified if they meet at least one of the following criteria: (1) the test name or directory path includes the term *"load"*, or (2) the workload conditions do not match those of smoke or stress tests.

The first two authors of the paper independently label the tests based on these guidelines. In problematic cases of labeling, the authors open a discussion revolving around the guidelines described above to find an agreement. In case of disagreement, the third author (a postdoc researcher with three years of experience in the field), is involved, and the decision is made based on majority voting. The classification provides insight into the types of performance testing activities commonly adopted in open-source web applications. It also sheds light on how performance testing is interpreted in practice, possibly revealing cases in which the nomenclature is not adequately used.

B. Analysis of the Results

Figure 2 shows the adoption of performance testing over time. Although LOCUST and APACHE JMETER were released in 2004 and 2011 respectively, the most significant growth in adoption occurred between 2018 and 2022. Notably, many repositories integrated performance testing years after their creation. For instance, repositories created in 2016 adopted performance testing at various points between 2017 and 2023 rather than immediately.

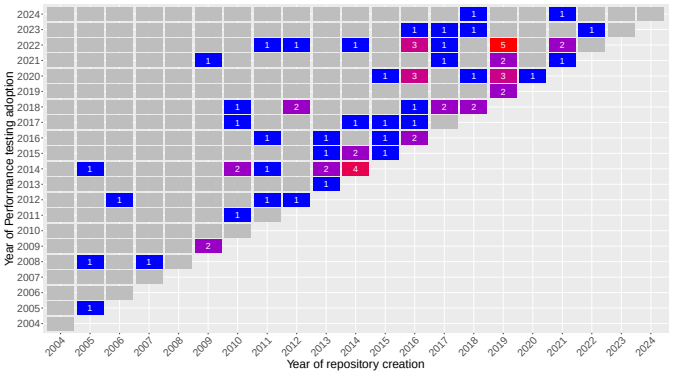

Figure 2: Repository creation and performance testing adoption over time. Each square shows the number of repositories that were created in the X-axis year and that adopted performance testing in the Y-axis year.

This trend suggests that performance testing is often introduced later in the development lifecycle, rather than being a priority from the start. To better understand this, we categorize adoption timing into three patterns:

- **Immediate adoption**: 23.38% of repositories integrated performance testing in the same year they were created.

- **Short-term adoption**: 42.86% adopted it within three years of creation.
- **Long-term adoption**: 33.77% incorporated it more than three years after creation.

These findings indicate that while some projects prioritize performance testing early on, a significant number introduces it only as their systems mature. This may suggest that performance concerns become more pressing as applications scale and experience real-world usage.

Figure 3 shows the distribution of three key performance test characteristics, namely the number of concurrent users, the number of network requests made, and the number of user behaviors defined. The violin plots illustrate the density and spread of values observed across the performance tests analyzed in our study, providing insights into common trends. The analysis highlights that the number of user behaviors per performance test exhibits a highly skewed distribution, with most tests (190) focusing on a single user behavior. This suggests that testers typically evaluate isolated scenarios rather than modeling multi-user interactions. Only 28 tests incorporate two behaviors, and only eight simulate more complex workloads. This suggests that performance tests in open-source projects are often simplified, rarely reflecting real-world multi-user workflows.

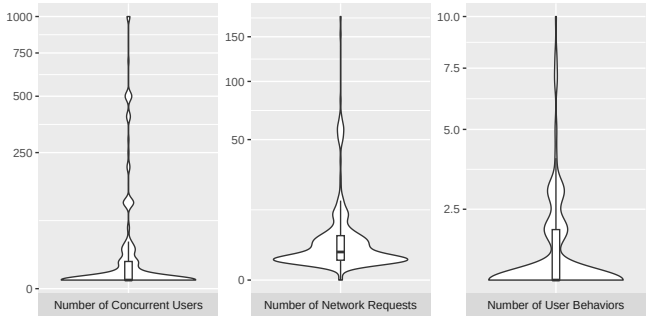

Figure 3: Characterization of performance test workloads. Note that the Y-axis has been transformed using `sqrt` to improve the visualization of data distribution.

Beyond modeling user behaviors, most tests also constrain the number of network requests per behavior. The majority (147 tests) include only a single request, while 33, 27, and 30 tests execute two, three, and four requests, respectively. Only 41 tests model more complex interactions of over 10 requests, with rare outliers exceeding 50 requests. A qualitative analysis reveals a common practice of validating multiple API endpoints within a single test using various inputs and parameters, rather than splitting them into structured test cases. As discussed in Section II, the low adoption of multiple user behaviors and sequential requests suggests that testers design performance tests to target isolated endpoints rather than simulate real-world workloads. This is further reinforced when examining concurrent users and test duration. The most frequent scenario (71 tests) involves a single user, indicating a focus on handling individual requests rather than system scalability; 55 tests scale between 2 and 10 users, but higher concurrency levels are rare, only 33 tests involve 11 to 100 users, and just 7 simulate over 1,000 users.

Looking at the test duration (Figure 4), a large percentage (84 tests) runs between 1 and 10 iterations, with 80 executing only once, while 45 define loops of between 10 and 1,000 iterations. Only 23 tests run indefinitely, while explicit duration definitions are rare. In particular, 10 tests specify duration in seconds between 20 and 425 seconds.

These findings highlight a key distinction between performance testing in open-source projects and industry settings [4], [5], [18]. In the past, one of the main challenges in performance testing was the time-consuming nature of the process and the massive amount of data generated, which required extensive analysis. However, this issue appears less relevant in open-source web applications, where the short execution times suggest that testers focus more on immediate response validation than on long-term system performance or stability. This aligns with the observation that most tests are designed to verify response status and response time, rather than assess broader system behavior under sustained load. As shown in Figure 5, the plugins and tools used in these tests primarily measure response times and request outcomes, rather than tracking key performance metrics such as CPU usage, memory consumption, or resource contention over time.

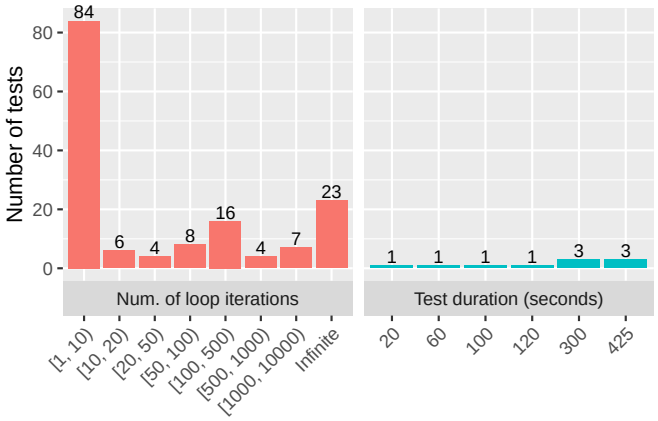

Figure 4: Characterization of performance test duration.

Concerning the different types of performance tests, load tests (focused on expected load handling) are with 110 instances the most common, followed by smoke tests (designed for basic validation) with 67 instances. Stress testing is less frequent, with only 26 instances, while we were not able to classify 22 LOCUST tests due to the absence of configuration details. More importantly, we identified some classification mistakes made by the test engineers themselves: of 9 tests named as a load test, 7 were actually smoke tests and 2 were actually stress tests. This suggests a lack of knowledge and standardized nomenclature regarding performance testing.

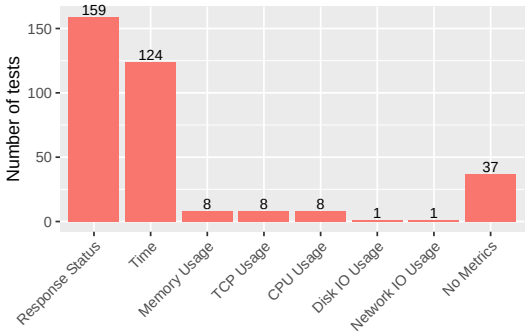

Figure 5: Characterization of the metrics considered in performance tests.

> **RQ$_1$.** Open-source web projects are slow adopters of performance testing, with growth appearing after 2018. Tests are simple, focusing on single-user behaviors (190 tests) and minimal requests (147 tests). Concurrency is low, and most tests (80) run a single iteration. Unlike industry, open-source projects prioritize response validation over system stability. Load tests are the most common, followed by smoke tests. Finally, the misclassifications suggest a lack of standardized performance testing practices.

V. RQ$_2$: What are the main characteristics of projects adopting Performance Testing?

A. Research Method

To answer **RQ$_2$**, we analyze whether projects that adopted performance testing exhibit distinct software repository characteristics. Understanding whether there are some differences is crucial, as it can reveal patterns and attributes of the repository that encourage or support the integration of such practices. We conduct a statistical analysis to explore the characteristics of projects adopting performance testing. Specifically, we investigate three dimensions. First, we analyze the **project maturity** by computing the lines of code (LOC), project age, and the number of tests (excluding performance tests) to understand the scale and testing emphasis of the projects. We also consider the programming languages used, as it can influence the availability and adoption of testing tools. Next, we analyze the **popularity**, including the number of stars and watchers, which reflect community interest and engagement. We hypothesize that more popular projects might adopt performance testing to ensure reliability and maintain user satisfaction. Finally, we explore the **development** state by assessing the number of distinct contributors, commits, and total issues per project. A larger number of contributors or more frequent commits could be indicative of a more dynamic development environment, potentially leading to increased adoption of performance testing to maintain system stability and performance.

The analysis is performed on all the 5,563 non-trivial web application repositories. Specifically, our sample includes the 77 repositories with performance testing and 5,486 without performance testing. To compute the metrics, we use SEART [36] and PyDriller [37]. Once the metrics are computed, we assess whether there are differences between the two sets. We analyze the metric distributions using boxplots and apply the Kolmogorov-Smirnov test [38] to check for normality. Then, we use the non-parametric Mann-Whitney U test [39] to determine if there are statistically significant differences. We set the confidence level α to 0.05, and account for multiple comparisons by applying the Bonferroni correction [40]. When statistically significant differences are detected, we compute the effect size using Cliff's delta [38], [41]. Finally, we build a Logistic Regression Model to determine if the statistically significant metrics remain influential when considered together. We use the `glm` function in R to implement the model, ensuring no multi-collinearity by applying the `vif` (Variance Inflation Factors) function with a threshold value of 5 [42].

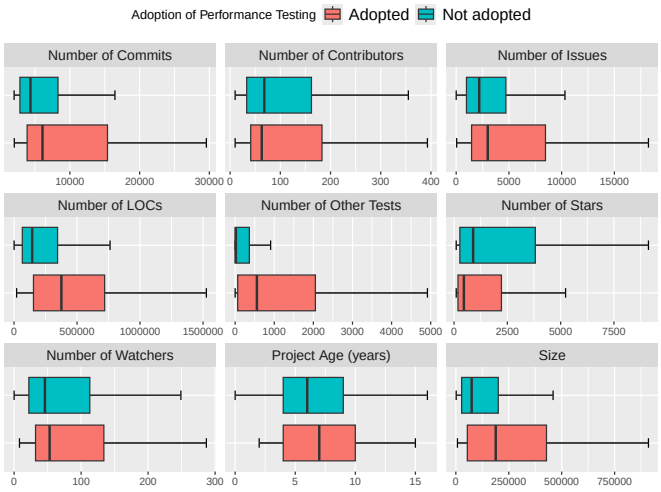

Figure 6: Distribution of repository characteristics for projects with and without performance tests. Outliers are hidden for ease of visualization.

B. Analysis of the Results

The distribution of the considered characteristics across the two groups of projects is shown in Figure 6. The boxplots highlight key differences between projects with and without performance tests. Specifically, projects with performance tests tend to be larger, both in terms of lines of code (LOC) and size, which suggests that as project complexity increases, adoption becomes more common. Furthermore, repositories that have already implemented other testing practices are more likely to incorporate performance testing, which could be indicative of a more comprehensive quality assurance (QA) process. This is further reflected in the higher number of issues reported, suggesting more thorough issue tracking and resolution. Furthermore, the adoption of performance testing is more common in older repositories, with a higher number of commits, which points to a more mature development process.

Table II: Results of the Mann-Whitney U and Cliff's delta Statistical Tests. The *p-values* in **bold** refer to the metrics for which we accepted the alternative hypothesis, i.e., the distribution of the characteristics in the two groups differs in a statistically significant way.

	Mann-Whitney U Test	Effect size	
Characteristic	**p-value**	**Cliff's delta**	**Interpret.**
# of LOCs	$\mathbf{5.03 \times 10^{-8}}$	0.36	Medium
# of Contributors	0.59	-	-
# of Stars	**0.01**	-0.15	Small
# of Commits	$\mathbf{8.3 \times 10^{-5}}$	0.26	Small
Project Age (years)	0.18	-	-
# of Watchers	**0.08**	0.13	Small
Size	$\mathbf{1.59 \times 10^{-5}}$	0.30	Medium
# of Other Tests	$\mathbf{2.71 \times 10^{-12}}$	0.47	Large
# of Issues	**0.0025**	0.19	Small

Table II reports on the results of the statistical tests and their effect sizes. Highlighted in bold are the *p-values* for which we accepted the alternative hypothesis; i.e., the distribution of the characteristic differs in a statistically significant way between the two groups. The analysis reveals statistically significant differences in most characteristics, except for the project age and the number of contributors. The effect sizes range from small to medium, with LOC and project size exhibiting medium effect, confirming that projects adopting performance testing are generally larger. Notably, the number of other tests is the only characteristic with a large effect size, reinforcing the idea that these projects have more solid and comprehensive testing activities. However, project age and number of contributors do not show significant differences, suggesting that these factors have a weaker correlation with the adoption of performance testing.

Finally, Table III reports the results of the *Logistic Regression Model*, which highlights that some characteristics could be used to predict the likelihood of a project adopting performance testing. Specifically, Total Issues, LOC, size, IsWebJava, and IsWebPy (i.e., the programming language adopted in the web application) show significant positive associations with performance test adoption. When considering all features together, the number of commits shows a significant negative association. Other factors, such as contributors, project age, watchers, and other test types, do not show statistically significant effects, suggesting that these aspects have a limited influence on the likelihood of adopting performance testing.

> **RQ$_2$.** Projects with performance tests tend to have more LOC, larger size, and a higher number of tests, suggesting a strong commitment to quality assurance. Statistical analysis confirms significant differences in most characteristics, except for project age and contributors. The logistic regression model highlights that LOC, size, total issues, and programming languages adoption such as JAVA or PYTHON are strong predictors of performance test adoption.

Table III: Results of the Logistic Regression Model. For each variable, the table reports the estimate, standard error, and statistical significance. Statistical significance is denoted by: '***' (p<0.001), '**' (p<0.01), '*' (p<0.05).

	Logistic Regression Coefficients			
	Estimate	**S.E**	**Sig.**	**Rel.**
(Intercept)	-5.18	0.42	***	↘
LOC	3.23	1.53	*	↗
Contributors	0.87	2.98		-
Stars	-9.87	10.91		-
Commits	-11.13	5.73	.	↘
Project Age	0.10	0.59		-
Watchers	-1.52	7.45		-
size	9.08	3.73	*	↗
OtherTest	0.63	8.04		-
Total Issue	11.48	4.45	**	↗
IsWebJava	2.28	0.29	***	↗
IsWebJs	-0.41	0.39		-
IsWebTs	0.28	0.34		-
IsWebPy	1.08	0.34	***	↗

VI. RQ$_3$: HOW FREQUENTLY ARE PERFORMANCE TESTS CHANGED?

A. Research Method

Inspired by previous research on the prevalence and maintenance of functional GUI test [33], we investigate the evolution of performance tests as the web application under test changes over time. To this end, we analyze the commit activities related to both performance tests and the application itself (as shown at the top of Table IV), quantifying how frequently performance tests are modified, added, or removed.

We further assess the lifespan and maintenance patterns of individual performance test files. We define the lifespan of a performance test file as a sequence of similar blobs across successive commits. Specifically, the first blob represents the initial introduction of a test file; the subsequent blobs correspond to modifications of the test file while retaining similar content, and finally, the last blob marks the file's final occurrence. To determine whether two blobs from consecutive commits are similar, we apply the definition provided by Christophe et al. [33], considering two blobs as similar when the more recent one retains at least 66% of the lines from the older one. When this threshold is not met, the file is considered replaced rather than modified. These definitions result in the following maintenance metrics related to a single performance test file, shown at the bottom of Table IV.

To collect the metrics previously described, we conduct the historical analysis through the use of PYDRILLER and collect all the versions of performance test files by excluding instances where tests were added, renamed, or removed.

Table IV: Definitions of performance test maintenance metrics. Note that the bottom section of the table presents metrics related to the individual lifecycle of a test.

Metric	Description
PTC	Refers to commits that add, modify, or delete at least one performance test.
AC	Refers to commits that add, modify, or delete web application code without touching performance test files.
SV_{Day}	Number of days the performance test survived.
SV_{Mod}	Number of modifications to the performance test.
SV_{AC}	Number of application commits the test survived.
SV_{PTC}	Number of performance test commits the test survived.

B. Analysis of the Results

Table V provides a summary of metrics related to the commit activity. The mean of the number of commits that involve performance tests (#PTC) is 6.68 with a relatively high standard deviation of 8.07, indicating variability in how frequently performance tests are modified across projects. The median and the first quartile of 3 and 2 respectively suggest that a substantial portion of projects have a low number of performance test-related commits. In contrast, there are projects with a higher frequency of performance test modifications, as indicated by the maximum of 37. Looking at the application commits (#AC), the mean is 1,771.87 with a standard deviation of 4,605.05, suggesting that application code is modified more frequently than performance tests. The differences between #AC and #PTC suggest that performance tests are not updated as often as application code, potentially leading to outdated or inadequate test coverage over time.

Table V: Statistics of co-evolution and maintenance metrics.

	Co-evolution metrics		Maintenance metrics			
Statistic	**#PTC**	**#AC**	**SV_{Day}**	**SV_{Mod}**	**SV_{AC}**	**SV_{PTC}**
Mean	6.68	1,771.87	1,787.70	2.90	2,531.95	6.46
Std Deviation	8.07	4,605.05	1,464.58	2.87	4,428.38	6.67
Median	3	429	1,394	2	1,345	4
1st Quartile	2	193.75	550	1	104	2
3rd Quartile	8.25	727.25	2,908	3	2,348	9
Min	1	10	1	1	1	1
Max	37	24,331	6,347	26	24,330	37

Regarding maintenance metrics, performance tests generally persist for long periods. The survival duration (SV_{Day}) has a mean of 1,787.70 days and a median of 1,394 days, indicating long-term retention. However, the high standard deviation (1,464.58) reveals substantial variation—while some tests last only a single day, others remain for up to 17 years (6,347 days). The modification frequency per test file (SV_{Mod}) is relatively low, with a median of two and a mean of 2.90, suggesting that most tests undergo minimal updates over time. Notably, the first quartile and minimum value are both 1, meaning that a significant portion of tests are never modified after creation, raising concerns about their long-term relevance. However, a few tests are updated frequently, with some modified up to 26 times. This trend is further confirmed when considering application changes. The SV_{AC} metric shows a mean of 2,531.95 and a median of 1,345, indicating that performance tests generally persist across multiple application updates. Yet, the wide range (1 to 24,330) suggests that while some tests remain relevant through extensive evolution, many become obsolete or are abandoned quickly. Similarly, SV_{PTC} reveals that performance tests survive through a mean of 6.46 and a median of four test-related commits, reinforcing the observation that most are seldom updated, though a few undergo frequent modifications over time.

> **RQ_3.** Performance tests are generally long-lived, with a median survival of 1,394 days. Most tests undergo minimal modifications, and many are never updated after creation, raising concerns about long-term relevance. While some tests persist across multiple application updates (median of 1,345 commits), others become obsolete quickly. Performance tests are modified infrequently compared to application code, suggesting a risk of outdated test coverage. However, a subset of tests undergoes frequent updates, indicating that maintenance needs vary significantly across projects.

VII. RQ_4: How are performance tests modified?

A. Research Method

In **RQ_3**, we observed significant variations in maintenance activities across projects. To gain deeper insights into when and why these modifications occur, we analyze the evolution of performance tests over time. Among the 226 tests examined, only 120 have at least one subsequent commit reflecting changes. For these 120 tests, we investigate 467 commits to understand the motivations behind the modifications. To achieve this, we employ *qualitative content analysis* [43], a research method in which one or more inspectors systematically examine the data to infer meaning and identify emerging concepts. Specifically, the first and third authors manually review each test version, labeling the modifications between successive versions. Whenever a change does not fit within the existing categories, a new label is introduced to capture its nature accurately.

B. Analysis of the Results

Through the analysis of 787 individual modifications (several commits contained grouped multiple changes), we identified 192 unique labels. These labels represent the basis for a taxonomy of performance test changes, as illustrated in Figure 7. We found ten main categories of modifications, each capturing a distinct aspect of how the test evolved. Three categories exhibited finer granularity and were further split into sub-categories to account for specific variations in modification patterns. In one of these cases, an additional sub-level classification was introduced to distinguish between nuanced forms of changes.

The most common category is *Workload Shape Modifications* (298 cases), which encompasses changes aimed at

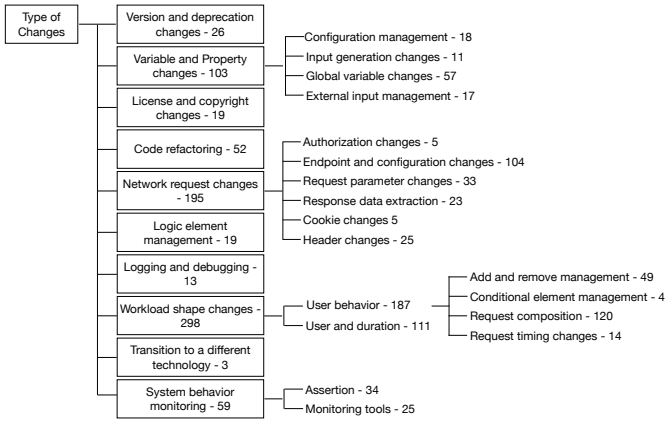

Figure 7: The taxonomy with the different categories of changes: the number represents the number of times that specific change was detected in the analyzed commits.

adjusting test duration, user load, and request structure. These modifications primarily fall into two sub-categories: *User behavior* modifications and *User and duration* adjustments. Developers frequently fine-tune workload parameters by altering test duration, loop iterations, and user concurrency to optimize load intensity and simulation accuracy. Additionally, they modify user behavior composition by enabling, disabling, adding, or removing user actions. Request *timing changes* (14) such as adjusting delays, timeouts, and pacing between requests are also common. The second most prevalent category, *Network Request* changes (195 cases), reflects the continuous adaptation of test configurations to evolving API requirements. A substantial portion of changes involves updating *Endpoints and Configurations*, including modifications to URL paths, query parameters, ports, and hosts. Many changes also affect POST parameters and authentication mechanisms, such as adjusting access tokens and authorization headers. Additionally, testers frequently modify response data extraction logic, updating `JSONPath` extractors or introducing new processors to ensure accurate test execution. Other changes that we find in this category are *Request Parameter*, *Header*, *Data Extraction Adjustments*, *Cookie Management*, and *Authorization*.

The *Variable and Property* category (103 cases) encompasses adjustments to global and test-specific variables. We can observe changes to *Global Variable*, *Configuration Management*, *External Input Management*, and *Input Generation Adjustments*. Developers frequently introduce user-defined variables, which have values that are updated dynamically, and replace static values with dynamically generated ones to improve test flexibility. Changes also refer to input sources, such as modifying file paths and delimiters or incorporating random input generators. Additionally, system parameters like timeouts, command-line arguments, and resource allocation settings are often fine-tuned.

System Behavior Monitoring modifications (59) reinforce findings from RQ_1, where we observed that testers prioritize endpoint reliability over detailed performance metrics. The two main sub-categories are *Assertion Changes*, which involve modifications to validation checks ensuring system responses meet expected conditions, and *Monitoring Tools Adjustments*, where testers integrate or refine external monitoring tools to track system performance and stability.

The *Code Refactoring* (52 occurrences) category of changes aims at improving maintainability without altering test functionality. These changes include formatting adjustments, import updates, and variable renaming, reflecting a strong emphasis on readability and reducing technical debt over time.

Other categories, though less frequent, capture additional aspects of performance test evolution. *Version and Deprecation Updates* (26) ensure compatibility by upgrading or downgrading dependencies such as APACHE JMETER, replacing deprecated functions, and adopting new APIs. *License and Copyright* changes (19) involve updates to copyright statements and license headers. *Logic Element Management* (19 cases) and *Technology Transitions* (3 cases) reflect structural test modifications. The former includes modifications to scripting and pre-processing elements, such as introducing or removing `BeanShellPreProcessor` and `JSR223Sampler` scripts, suggesting that built-in tools were insufficient, leading testers to rely on custom scripting. The latter, though rare, involve replacing core test elements, such as switching from `TaskSet` to `SequentialTaskSet` in LOCUST or replacing a `Thread Group` with an `Ultimate Thread Group` in APACHE JMETER. Finally, *Logging and Debugging* modifications (13) aim to enhance error handling and traceability, including introducing logging tools, adjusting configurations, and adding print statements for debugging.

> **RQ_4.** The taxonomy shows that performance tests evolve through a variety of modifications, with most involving *Workload Shape* (298 cases) and *Network Request* (195 cases) updates. Other key modifications include *Variable Adjustments* (103), *Monitoring* (59), and *Refactoring* (52). Less frequent updates address compatibility, structure, and debugging. Finally, we also observed changes to update license and copyright statements.

VIII. DISCUSSION AND IMPLICATIONS

The results provide several observations, reflections, and implications for research and practice.

For *practitioners*, the study highlights that testers often validate multiple API endpoints within a single test or enable/disable user behavior within the same test to simulate different workloads, rather than using different test files. These bad practices lead to misclassification of performance test activities, highlighting a lack of standardized practices, and underlining the need for naming conventions and structured documentation to enhance the organization of performance tests. Since many tests remain unchanged after their initial creation, implementing automated monitoring and update mechanisms could help maintain their accuracy as software evolves, ensuring that tests remain relevant and reflective of real-world

workloads. Additionally, our taxonomy provides a structured perspective on test evolution, revealing that most modifications focus on workload shape, network request adjustments, and assertion refinements. This suggests that practitioners should prioritize maintaining test realism and adaptability by regularly reviewing workload configurations and endpoint interactions to align with evolving system behavior.

For *researchers*, our findings indicate that performance tests often lag behind application updates, leading to potential inconsistencies in software validation. Investigating automated approaches to synchronize test evolution with system changes could enhance test reliability and effectiveness. The high frequency of network request modifications suggests that performance tests are particularly sensitive to API evolution, making it crucial to explore techniques for automating API-aware test updates. Moreover, our taxonomy serves as a foundation for developing automated classification techniques that identify and categorize test modifications, enabling researchers to build predictive models for anticipating maintenance needs and optimizing test co-evolution strategies. Beyond classification, the taxonomy also presents an opportunity to deepen our understanding of test evolution. While some tests persist with minimal updates, others undergo frequent modifications, raising important questions about test relevance, maintenance costs, and the factors influencing test longevity. Further research could investigate whether specific modification patterns correlate with test effectiveness and how they impact software quality over time.

IX. Threats to Validity

Several factors may have influenced the conclusions drawn in this study. Below, we discuss the main limitations and mitigation strategies applied [44].

Threats to Construct Validity. Our study relied on GitHub web applications that use Apache JMeter and Locust for performance testing activities. Although other common or paid frameworks may have been excluded , we believe that our analysis accurately represents the current state of performance testing practices.

As for the metrics computed in $\mathbf{RQ}_2$, we relied on automatic tools to extract project characteristics related to maturity, popularity, and development activity. We acknowledge the potential noise that may arise from automated extraction, such as inaccuracies in LOC calculations, incomplete metadata, or limitations in detecting development contributions. To partially mitigate this threat, we employed well-established tools that have been previously evaluated, demonstrating good accuracy in repository mining and software evolution analysis [36], [37].

Threats to Conclusion Validity. The first threat concerns potential research bias during coding in $\mathbf{RQ}_4$, since this was based on the researchers' judgment. Although we adopted preventive measures like independent review, we cannot guarantee that coding would have been conducted differently by other researchers. However, despite this limitation, we believe our taxonomy remains a valuable contribution, as no existing classification of this kind is currently available in performance testing research.

Similarly, the classification of performance testing activities and the workload design in $\mathbf{RQ}_1$ rely on manual analysis conducted by two authors of the paper, both with three years of experience in the field. While manual analysis introduces the potential for subjectivity, the authors' expertise partially mitigate the risk of inconsistent classifications. Nonetheless, variations in interpretation may still occur, particularly when identifying performance activities and analyzing workload design. In this respect, we made all our data publicly available to make our results repeatable and reproducible [8].

Additionally, in the classification of performance testing activities, we were unable to classify 22 Locust tests due to the absence of configuration details. While this could limit the completeness of the analysis, we believe that the remaining data still provides valuable insights into performance testing activities in open-source web applications.

As for the statistical methods employed in $\mathbf{RQ}_2$, we selected the Logistic Regression Model after verifying its suitability for our purpose. In addition, we applied the `vif` to discard non-relevant metrics. These procedures followed established guidelines [42], making us confident of the validity of the conclusions drawn.

Threats to External Validity. Our study focused on open-source projects selected from GitHub, which are only a fraction of the complete picture of open-source software. Therefore, we cannot ensure that our findings generalize when considering different software systems or different contexts, e.g., industry projects. In this regard, we released all materials publicly available to stimulate further research that may corroborate our findings in different contexts [8].

X. Conclusion

This study explored the adoption and maintenance of performance testing in open-source projects. Analyzing 77 projects using Apache JMeter and Locust, we found that performance testing is often introduced late and primarily targets isolated endpoint validation rather than comprehensive workload simulations. While these tests tend to be long-lived, they are rarely updated, raising concerns about their long-term relevance. Our analysis on 787 individual modifications reveals that the most frequent maintenance activities involve workload shape adjustments, network request modifications, and assertion refinements. These findings underscore the need for improved workload modeling, structured test maintenance, and automated adaptation to evolving APIs. By providing a taxonomy of performance test changes, this study lays the groundwork for enhancing performance testing practices in open-source software development. As part of our future agenda, we plan to investigate strategies to improve test maintainability and investigate whether integrating automated approaches for workload modeling.

REFERENCES

[1] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[2] K. Eaton, "How one second could cost amazon $1.6 billion in sales," 2012, last accessed: Oct. 24, 2022. [Online]. Available: https://web.archive.org/web/20221006004855/https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[3] D. A. Menascé, "Load testing of web sites," *IEEE internet computing*, vol. 6, no. 4, pp. 70–74, 2002.

[4] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 243–252.

[5] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[6] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 373–384. [Online]. Available: https://doi.org/10.1145/3030207.3030213

[7] S. Di Meglio, L. L. L. Starace, V. Pontillo, R. Opdebeeck, C. De Roover, and S. Di Martino, "E2egit: A dataset of end-to-end web tests in open source projects," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2025, pp. 10–15.

[8] "Characterizing performance testing in open-source web projects: Adoption, practices, and maintenance — figshare.com," https://figshare.com/s/8d8fc0045e0a627ade0e, 2025.

[9] A. van Hoorn, M. Rohr, and W. Hasselbring, "Generating probabilistic and intensity-varying workload for web-based software systems," in *Performance Evaluation: Metrics, Models and Benchmarks*, S. Kounev, I. Gorton, and K. Sachs, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 124–143.

[10] K. Yorkston, *Performance Testing Tasks*. Berkeley, CA: Apress, 2021, pp. 195–354. [Online]. Available: https://doi.org/10.1007/978-1-4842-7255-8_4

[11] S. Di Meglio, L. L. L. Starace, and S. Di Martino, "Starting a new rest api project? a performance benchmark of frameworks and execution environments." in *IWSM-Mensura*, 2023.

[12] R. P., K. Bhargav, and M. Tech, "A survey on performance testing approaches of web application and importance of wan simulation in performance testing," *International Journal on Computer Science and Engineering*, vol. 4, 05 2012.

[13] K. M. Alsante, L. Martin, and S. W. Baertschi, "A stress testing benchmarking study," *Pharmaceutical technology*, vol. 27, no. 2, pp. 60–73, 2003.

[14] M. Kalita and T. Bezboruah, "Investigation on performance testing and evaluation of prewebd: A. net technique for implementing web application," *IET software*, vol. 5, no. 4, pp. 357–365, 2011.

[15] S. Nejati, S. Di Alesio, M. Sabetzadeh, and L. Briand, "Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing," in *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*. Springer, 2012, pp. 759–775.

[16] S. Di Meglio and L. L. L. Starace, "Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies," *IEEE Access*, 2024.

[17] C. Cannavacciuolo and L. Mariani, "Smoke testing of cloud systems," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 47–57.

[18] E. Battista, S. Di Martino, S. Di Meglio, F. Scippacercola, and L. L. L. Starace, "E2e-loader: A framework to support performance testing of web applications," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 351–361.

[19] D. Krishnamurthy, J. A. Rolia, and S. Majumdar, "Swat: A tool for stress testing session-based web applications." Citeseer.

[20] S. Shrivastava and S. Prapulla, "Comprehensive review of load testing tools," *International Research Journal of Engineering and Technology*, vol. 7, no. 3392-3395, p. 43, 2020.

[21] [Online]. Available: https://jmeter.apache.org/

[22] R. K. Lenka, M. R. Dey, P. Bhanse, and R. K. Barik, "Performance and load testing: Tools and challenges," in *2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)*. IEEE, 2018, pp. 2257–2261.

[23] R. Abbas, Z. Sultan, and S. N. Bhatti, "Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege," in *2017 International Conference on Communication Technologies (ComTech)*, 2017, pp. 39–44.

[24] "Locust.io — locust.io," https://locust.io/, [Accessed 23-02-2025].

[25] H. Malik, B. Adams, and A. E. Hassan, "Pinpointing the subsystems responsible for the performance deviations in a load test," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 201–210.

[26] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE transactions on software engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[27] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *Proceedings of the 3rd International Workshop on Software and Performance*, 2002, pp. 17–24.

[28] L. Traini, "Exploring performance assurance practices and challenges in agile software development: an ethnographic study," *Empirical Software Engineering*, vol. 27, no. 3, p. 74, 2022.

[29] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st International Conference on software testing, verification, and validation*. IEEE, 2008, pp. 220–229.

[30] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, pp. 325–364, 2011.

[31] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[32] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 151–154.

[33] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 141–150.

[34] "Smoke testing: A beginner's guide — Grafana Labs — grafana.com," https://grafana.com/blog/2024/01/30/smoke-testing/, [Accessed 26-02-2025].

[35] V. K. Chauhan, "Smoke testing."

[36] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 560–564.

[37] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2018, p. 908–911.

[38] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, pp. 579–630, 2017.

[39] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[40] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[41] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[42] R. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[43] S. Cavanagh, "Content analysis: concepts, methods and applications." *Nurse researcher*, vol. 4, no. 3, pp. 5–16, 1997.

[44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.