CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Analyzing Java Microbenchmark Harness (JMH) Performance in Open-Source Systems

Bachelor of Science Thesis in Software Engineering and Management

Ayah Miqdad

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The Author grants to University of Gothenburg and Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let University of Gothenburg and Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Performance Measurement Practices in Open-Source Java Projects
An In-Depth Analysis of Microbenchmarking, Stability, and Variability Across 586 Microbenchmarks from 30 Projects
These options effectively convey the main content and scope of the study.

© Ayah Miqdad, May 2024.

Supervisor: Philipp Leitner
Examiner: Daniel Strüber

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: Illustration inspired by "Benchmark Java Applications using JMH" by Rahul (December 20, 2019, [13])]

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024

Analyzing Java Microbenchmark Harness (JMH) Performance in Open-Source Systems

Ayah Miqdad
Software Engineering and Management
University of Gothenburg
Sweden
gusabuay@student.gu.se

Abstract
Microbenchmarking is widely used in Java software testing, especially for open-source projects. This study examines performance measurement practices in open-source Java projects, analyzing 586 microbenchmarks across 30 projects. We investigate performance test durations, stability, and variability, revealing insights through exploratory data analysis and statistical modeling. Our findings offer guidance for achieving steady-state performance and understanding performance measurement patterns. Additionally, we explore performance metric variability, emphasizing the impact of outliers. Our study contributes actionable insights for enhancing software quality and performance in open-source projects. We also employ density plots to visually represent performance metric variability. While acknowledging limitations, our research bridges theory and practice, empowering developers and researchers to optimize Java software performance in real-world scenarios.

Keywords *— microbenchmarking, performance testing, open source, Java projects, performance measurement, data analysis, statistical modeling, software quality*

1. Introduction

In software engineering, a performance measurement system (PMS) is a set of processes, tools, and metrics used to assess and quantify the performance of various aspects of the software development life cycle [1]. This system plays a vital role in assessing the performance of open-source systems (OSS), ensuring that goals for metrics like response time and resource utilization are consistently met [1]. OSS projects, renowned for fostering collaboration and innovation, are essential components of modern software development [1]. However, as their popularity and complexity increase, understanding and managing their performance becomes increasingly critical [2].

Despite the availability of performance measurement tools, the current state of performance assessment in OSS presents significant challenges [6]. Key obstacles include parameter selection for tests, establishing testing environments' steady states, and analyzing performance variations [6]. Moreover, as OSS project demand escalates, performance monitoring systems encounter difficulties handling large datasets, necessitating scalable solutions [1].

This study aims to illuminate the OSS performance assessment's complexities [7]. By analyzing performance measurement in 30 diverse OSS projects, our goal is to present an overview of current strategies and challenges in OSS performance analysis. Through exploratory data analysis and statistical modeling, we aim to uncover patterns and correlations within the dataset, offering actionable insights for improving performance measurement practices from the work of Mey et al. [3]. Our study aims to bridge the gap between theory and practice in OSS performance assessment, ultimately contributing to the improvement of software quality and performance in open-source projects.

2. Background

Microbenchmarking is a technique used in software performance evaluation to measure the execution time or other performance metrics of small code segments, typically at the method or function level. Unlike macrobenchmarks, which evaluate entire applications or systems under realistic conditions, microbenchmarks focus on specific code segments to isolate and analyze their performance characteristics [1].

The Java Microbenchmark Harness (JMH) is a widely adopted framework for conducting microbenchmarking in Java. Developed as part of the OpenJDK ecosystem, JMH provides developers with a robust and standardized platform to design, execute, and analyze microbenchmarks. It offers features such as parameterization, warmup iterations, and measurement iterations to ensure accurate and reliable performance measurements [2]. Steady-state performance, also known as stability, is a critical aspect of microbenchmarking. It refers to the point during benchmark execution where the performance metrics stabilize, indicating that the JVM (Java Virtual Machine) has reached an optimized state for executing the benchmarked code. Achieving steady-state performance is essential for obtaining consistent and meaningful performance measurements [3].

Java, like most programming languages, utilizes a virtual machine (VM) to execute code. The Java VM justifies time computation by loading and optimizing code at runtime. Constant values and frequently used code segments are specially placed in memory and optimized for faster execution. For instance, loops that are frequently executed are loaded into memory, allowing the JVM to optimize their execution by avoiding repeated loading of loop statements [4]. Furthermore, Java provides developers with the flexibility to configure the benchmarking system according to their requirements. This configurability allows developers to tailor the benchmarking process to specific use cases and performance evaluation scenarios, enhancing the accuracy and relevance of the benchmark results [5].

In the context of our research, we utilize a comprehensive dataset provided by Traini et al., which contains performance measurements from 586 microbenchmarks across 30 popular Java open-source projects [9]. These projects span various domains, including application servers, libraries, and databases. The benchmarks were meticulously executed using the JMH framework, ensuring the reliability and reproducibility of the performance measurements. The dataset includes information on Git revisions, allowing us to trace the version

of the projects at which the benchmarks were conducted [4]. By leveraging this rich dataset and exploring fundamental concepts such as microbenchmarking, JMH, and steady-state performance, we aim to gain insights into the performance measurement practices in open-source Java projects.

3. Related Work

As is typical these days, there is a lot of research in academia about performance measurement in open-source systems. Laaber and Leitner have shown the gradual importance of permanent evaluation in open-source software development, particularly in microbenchmarks and continuous integration [2]. Laaber and Leitner thoroughly investigated the quality of software microbenchmark suites across ten open-source projects written in Java and Go. Their study aimed to assess the suitability of these suites for continuous performance evaluation within CI environments. Moreover, they examined the variability of results across benchmark suites, noting that Go benchmarks generally exhibited lower variability than Java benchmarks. They also introduced a novel performance-test quality metric called ABS, which evaluated the benchmark suites' ability to detect slowdowns. The ABS scores varied significantly across the study subjects, ranging from 10% to 100%. Their study methodology and outcomes offer valuable insights applicable to evaluating microbenchmarking quality, improving existing benchmark suites, and selecting benchmarks for inclusion in CI processes. In the context of our research, Laaber and Philip's work underscores the importance of addressing benchmarking challenges and variability to enhance the accuracy and reliability of performance assessments in Java-based open-source projects. Their findings inform our understanding of the intricacies involved in microbenchmarking and provide valuable guidance for optimizing performance evaluation practices in real-world software development scenarios.

Moreover, McColl et al. contribute to the understanding of performance assessment in open-source systems, particularly with graph datasets, as a reference point for the challenges posed by large and irregular data [6]. This informs our understanding of data handling complexities, which guides our investigation of scalable systems and their function in handling a huge amount of data.

Additionally, Lazidis and teams' examination of open-source publish-subscribe systems provides important information on standardized features, design elements, and applications to loT and cloud computing [5]. Our research offers unique contributions that extend beyond their scope. Unlike their on-specific systems and applications to loT and cloud computing, our study aims to provide a broader understanding of performance measurement practices across a wide range of open-source projects. For example, our analysis revealed that the time taken for performance tests varied significantly among different projects, influenced by different factors such as specific values and parameters used in the testing process.

Barrett et al. researched virtual machine warmup, exploring its effects on performance. Their study revealed insights into the variability of warm-up behavior, highlighting the importance of understanding and mitigating warm-up effects in virtualized environments.

Although their work focuses on a different aspect of performance evaluation compared to our research, it underscores the complexity of performance assessment in modern computing systems. By acknowledging the challenges identified by Barrett et al., we contribute to a more comprehensive understanding of performance assessment in our specific domain of open-source systems [11].

Specifically, while their research focuses on established approaches to different designs, our work takes a more comprehensive approach, not only categorizing and analyzing different open-source projects but also looking into specific challenges of benchmarking variability and efficiency modeling with Java and Go libraries. Additionally, our research seeks to identify common patterns and best practices in performance measurement that can be applied across various domains and project types.

Furthermore, Lazidis and colleagues' work serves as a valuable reference point. When our research fills a gap by offering a practical solution and implementation strategies, it transforms into the unique challenges of performance measurement within open-source systems. Providing actionable insights empowers developers and contributors in the open-source community to optimize software performance effectively.

4. Research Methodology

This research aims to analyze the state of performance measurement in open-source systems, focusing on factors such as test duration, steady state achieving, and performance variability.

4.1 Research questions

The research questions guiding this study are as follows:

1. What is the average duration of performance tests in open-source systems?
2. How many test repetitions are needed to stabilize a benchmark's stability?
3. What is the range and magnitude of performance variability observed in open-source systems once a steady state is reached?

4.2 Data collection

The dataset provided by Traini et al. [9]. offers a comprehensive collection of performance measurements extracted from 586 microbenchmarks across 30 popular Java open-source projects. These projects encompass a wide range of domains, including application servers, libraries, and databases [9]. Each benchmark was carefully executed using the Java Microbenchmark Harness (JMH) framework, ensuring robust and reliable performance measurements. The dataset includes information on Git revisions, indicating the version of the projects at which the benchmarks were executed. The dataset collection tool is available on GitHub[1]

4.3 Data analysis

In this section, we describe the setup of our exploratory study and our methodology for analyzing the data. In addition, we will describe how to answer each research question.

[1] https://github.com/SEALABQualityGroup/icpe-data-challenge-jmh?tab=readme-ov-file

4.3.1 Exploratory Data Analysis (EDA)

EDA will be critical in understanding the dataset's characteristics and uncovering potential patterns and trends. We will utilize various visualization techniques tailored to microbenchmark data to gain deeper insights into performance measurement.

Data analysis will be done using the Python programming language and the Jupyter Notebook application. The advantage of using Python is the best programming language to learn to code. While Python boasts thousands of free and open-source libraries with statistical functions, its ease of use may vary among users. While some may find Python intuitive and straightforward to learn, others may encounter challenges, particularly if they are less experienced with programming languages [12]. The data analysis process involves several steps [10]. Firstly, Data Extraction: the process starts by getting the data, in our case, which is a file stored in JSON format from the dataset on GitHub that contains the JMH output. Secondly, Data cleaning: Once we have collected the data, we need to clean it, which helps identify and correct flows, mistakes, or conflicts in the dataset. So that ensures that the analysis is built on correct and dependable data, such as handling missing values (e.g., test durations, repetitions), which can otherwise lead to biased or incomplete analysis. To handle missing values, it usually uses the average (mean) or middle value (median) to fill in gaps. The user normally replaces missing values with the most common value (mode) for categories or types.

Thirdly, after data cleaning, we can begin data wrangling. This phase entails rearranging and reshaping the data to facilitate analysis, which may involve modifying fields, merging tables, and integrating data from multiple sources. The goal of this process is to prepare the data for the next stage. The process of analysis involves extracting patterns from the data that is now clean and in shape capturing trends or anomalies. Statistical analysis will be fundamental in this process. Finally, in the action step, it's time to do something with data analysis. We could be ready to build reports. communicate our results and support decision-making. The most significant libraries we will use are Pandas for data processing and Matplotlib and Seabourn for visualization. We efficiently identify patterns and detect anomalies through EDA, which guide our subsequent statistical analyses. This approach allows us to get meaningful conclusions from the data, ultimately contributing to a deeper understanding of performance measurement in open-source systems.

4.3.2 RQ1-Calculating the Average Duration

To calculate the average duration and standard deviation from the data stored in JSON files, the process involved the following steps:

- Reading data from a JSON file in the time-series folder. For each file, it extracted relevant information, such as the project name and benchmark name, from the title of the file. And extracted the measurement data.

- Calculating metrics for each benchmark: for each project, the script calculated the average duration and the standard deviation from the measurement data in the previous step.

- Saving data to a CSV file: to make the data accessible and printable, the script saved it to a CSV file. This file served as a structured format for storing the calculated metrics.

This will help in addressing the duration of performance measurement tests. To address this research question, we will analyze the duration of performance tests across the 586 microbenchmarks from 30 Java open-source projects provided by Traini et al. [9]. To identify common patterns, we complete descriptive statistics, namely the mean, median, and standard deviation of test durations. Additionally, we visualized the distribution of test durations among various projects using histograms or box plots to identify any trends or outliers.

4.3.3 RQ2- Achieving a Stable State

This research question focuses on repetition to achieve a steady state, and the study will investigate the number of repetitions required to complete a steady state in performance measurements. In the context of our research, a stable state refers to a point in the performance measurements where the values become consistent and show minimal fluctuations.
This indicates that the benchmark has reached a level of stability in its performance. When subsequent measurements provide similar results, thus providing reliable data for further analysis. Stability in performance measurements is crucial, as it ensures that initial setup conditions do not influence the results but rather reflect the true performance characteristics of the tested system. To achieve this, we employed a data visualization approach using Python. First, we gathered performance data from multiple JSON files, each representing a benchmarking run for a specific Java open-source project. We focused on the data from the first fork of each file to ensure consistency. Next, we extracted the performance measurements for the first 100 iterations of each benchmarking run. Limiting the analysis to the initial iterations allowed us to focus on the early stages of benchmark execution, where performance measurements are most likely to exhibit variability and instability.

Using the Matplotlib library in Python, we plotted the performance measurements against the iteration count for each benchmarking run. By visually inspecting these plots, we were able to identify patterns and trends in the performance data. Specifically, we looked for points at which the performance measurements stabilized, indicating the attainment of a steady state. To ensure accuracy in our analysis, we manually examined the plots to identify the exact iteration at which the performance measurements stabilized. This manual inspection allowed us to confirm the stability of the benchmarks and determine the point at which the steady state was reached.

4.3.4 RQ3- Measuring the Performance Variability

Finding the steady state, as established in RQ2, is essential for answering this research question. Subsequently, the focus shifts to quantifying the remaining variability at this stage. It is crucial to recognize that variance continues from the steady state to the end of the data for each benchmark. We used a statistical measurement such as mean, variance, range, and standard deviation for performance measurements across different benchmarks, specifically at the established steady

state. Finally, statistical analyses were conducted to quantify and characterize the observed variability while also identifying commonly used performance metrics, such as outliers.

Our analysis highlights outliers as a significant performance metric that affects variability. We observed that plots with a higher number of outliers tend to exhibit higher variance and variability, while those with fewer outliers demonstrate lower variance and variability. Recognizing outliers as performance metrics is crucial for understanding and interpreting the variability observed in performance testing data. This comprehensive approach enables the calculation of relevant statistical measures, providing insight into the scale and level of performance variability within open-source systems once a steady state is reached.

4.4 Methodological limitations

To examine the distribution of data and focus on the variability captured by standard deviations, it's important to acknowledge the limitations of using the standard deviation as a measure of variance. The Standard deviation assumes a normal distribution of data, which may not accurately represent the variability in real-world performance measurements. This becomes evident when observing Figures 10 and 11 in section 5. There are almost no data points in the lower green section, but many in the higher green part. Where the distribution deviates significantly from normality, leading to discrepancies between the data and the standard deviation-based analysis. In such cases, a more comprehensive measure, such as a bootstrapped confidence interval, may provide a more accurate representation of variability. However, it's worth noting that calculating a bootstrapped confidence interval can be time-consuming and may not always be feasible, depending on the scope of the analysis.

Despite these limitations, examining the data distribution and considering measures of variability like the standard deviation can provide valuable insights, particularly in identifying outliers or extreme values. Although common in real-life performance measurements, outliers should be carefully examined as they might affect the overall study findings and conclusions based on the data. Therefore, while the standard deviation provides a useful metric for understanding variability, it's essential to recognize its limitations and consider alternative measures when appropriate.

5. Results

The dataset comprised performance measurements extracted from 586 microbenchmarks across 30 popular Java open-source projects.

5.1 Average Duration

The analysis conducted to address RQ1 focused on determining the average duration of performance tests in open-source systems and identifying common patterns across different projects.

The descriptive statistics and visualizations reveal that the average duration of performance tests across various open-source projects exhibits considerable variability. This variability can be attributed to several factors, including the specific values and parameters used in the performance tests. Fig 1 and Fig 2 show the average duration for different benchmark projects. When the x-axis represents the benchmark project name that we observed from the name of the JSON files in the time-series data folder and the y-axis represents the mean average duration of each project. And then we split the plot into two different plots one with the highest duration and the other one with the lowest duration.

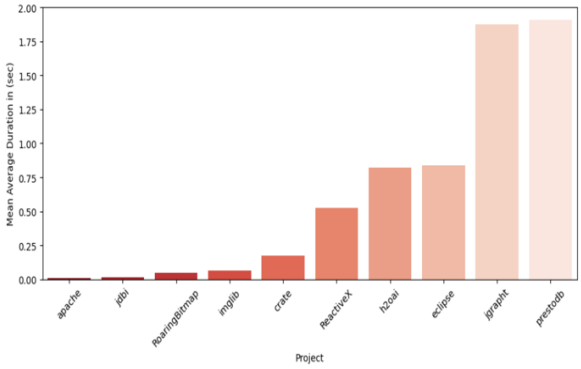

Fig 1: The projects with the highest duration

The first box plot in Fig 1, reveals that the "prestodb" project has the highest duration among all the projects included in the analysis. This finding suggests that the performance tests conducted within the "prestodb" project exhibit longer durations than other projects.

We observed the parameter from the name of the JSON files specially for project "prestodb. For example, "blockSize=1000000" indicates the size of data blocks processed during the benchmarking. A large block size increases the amount of data processed in each iteration. The parameter "nestedLevel=2" suggests the depth or the complexity of nested data structure within the benchmarks. In addition, the parameter "numberOfIteration=1000" indicates the number of times the benchmarking process is repeated. Performing many iterations can prolong the overall test duration, especially if each iteration involves significant computational tasks or data processing.

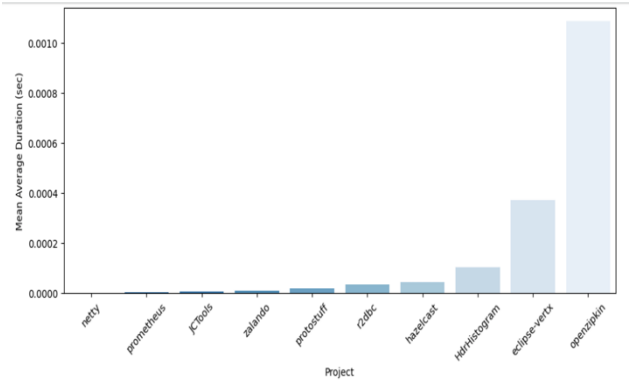

Fig 2: The projects with the lowest duration

The second box plot in Fig 2, reveals that the "netty" project has the lowest duration compared with the other projects included in the analysis. As observed the size of the data being processed in the benchmarking scenario such as "size=7" can impact the overall duration. Smaller data sizes lead to faster processing times compared to larger data sizes. Parameter "alignedCopyByteBuf=true" aligned copying byte buffer can contribute to efficient data handling and processing. Potentially, it will result in shorter durations. In addition, the

parameter “readonlyByteBuffer=false” indicates that the byte buffer is not read-only.

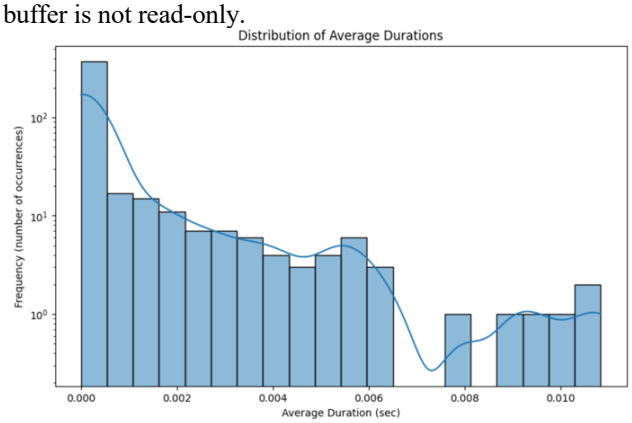

Fig 3: The Frequency of average duration, note that the y-axis is in logarithmic scale.

The frequency of performance test refers to the number of repetitions of a specific test duration within the dataset. By calculating how often each duration appears, we can construct a frequency distribution, which shows the distribution of test duration across the dataset.
This distribution provides valuable information about the characteristics of the testing process and helps to understand the overall performance and efficiency of the system under the test. The frequency of performance tests refers to how often specific test durations occur within the dataset. Constructing a frequency distribution helps visualize this distribution across the dataset. Typically, shorter benchmark execution times are more common than longer ones, as depicted in Fig.3. However, the figure shows the opposite trend, indicating longer durations occurring more frequently. This discrepancy suggests potential inconsistencies in the dataset or anomalies that warrant further investigation.

5.2 Achieving Steady State

After plotting all benchmark projects for the first fork, each containing 3000 measurement iterations, we focus on the initial 100 iterations for better visualization of changing points and assessing stability. In this scenario, we assume that if a benchmark reaches a steady state, it does so well before the 3000th iteration.

We manually observed the stable state of each project's plot. Looking for a point in the plot where the performance measurement stabilizes. This can be indicated by a relatively flat portion of the plot where fluctuations are minimal. For instance, in the benchmark distribution of data shown in Fig 4, we noticed that the line is initially high, then decreases. From our observation, we identified that the stable state in this plot begins from point 1, indicating that stability is achieved from the first iteration. In the context of steady-state behavior for the full dataset, the fluctuations at the line's rise and fall points, which are 6, 7, and 8, seem logical.

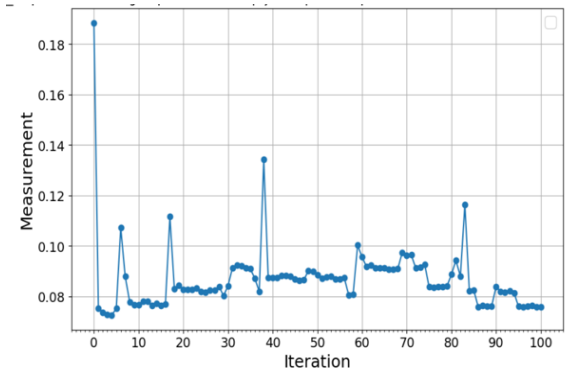

Fig 4: Distribution of benchmark project in (eclipse__eclipse-collections#org.eclipse.collections.impl.jmh.map.ChainMapPutTest.hppc#isPresized=false&loadFactor=0.75f&size=750000.json)

In Fig 5, we observed that the steady state begins from the fifth iteration, specifically from point 6 on the plot.

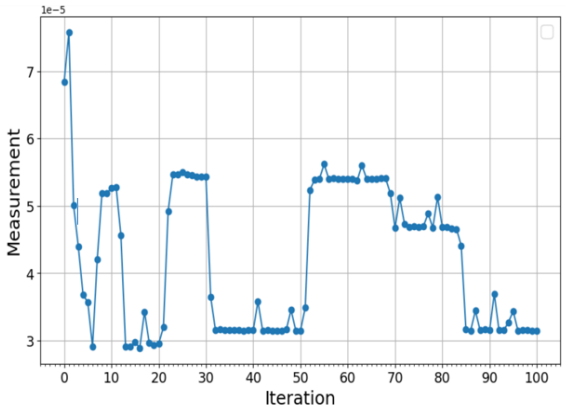

Fig 5: Distribution for benchmark project for (apache__camel#org.apache.camel.converter.ConverterBenchmarkTest.benchmarkConversionStringToChar#.json)

In Fig 6, it is evident that the steady state begins with the first iteration, as indicated by point 1. Subsequently, the measurements remain consistently stable without significant fluctuations during the duration of the plot.

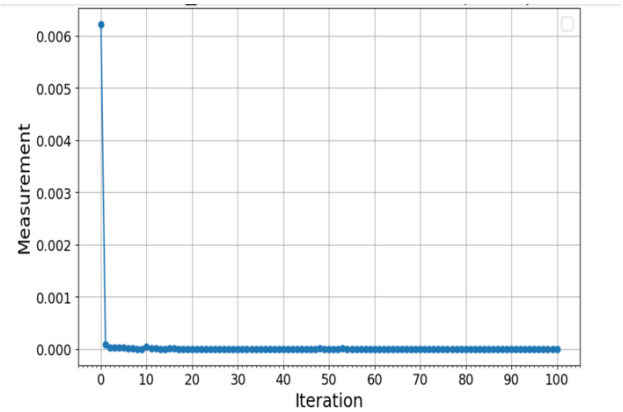

Fig 6: Distribution for the benchmark for (r2dbc__r2dbc-h2#io.r2dbc.h2.StagedResultSizeBenchmarks.parametrizedR2dbc#resultSize=10.json)

As observed in Fig 7, the steady state begins from the fourth iteration, which corresponds to point 3 on the plot, considering that the first iteration starts from zero.

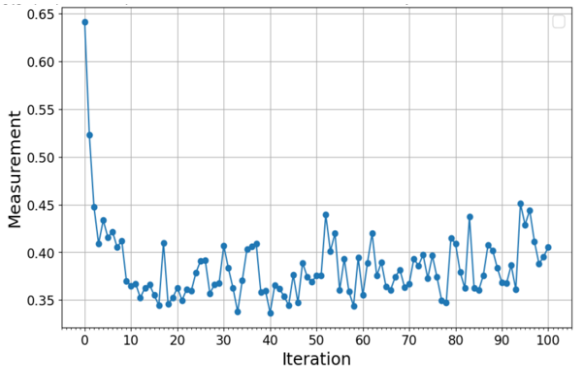

Fig 7: Distribution for benchmark project for (jgrapht__jgrapht#org.jgrapht.perf.shortestpath.KShortestPathsPerformance.testSuurballeKDisjointShortestPaths#k=50&n=100&numberOfQueries=20&p=0.3.json)

In Fig 8, the steady state for this benchmark commences from iteration five, indicating the sixth iteration as the starting point for stability.

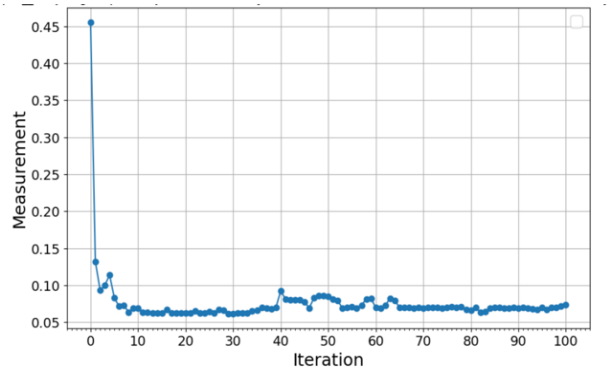

Fig 8: Distribution for the benchmark Project for (eclipse__rdf4j#org.eclipse.rdf4j.benchmark.QueryOrderBenchmark.selectDistinct#countk=50&limit=10&syncThreshold=0.json)

After visualizing the steady state or transition point for each benchmark, it became evident that all benchmarks typically reached a stable state within the initial 5 iterations. Only a small number of benchmarks required more than five iterations to reach stability.

As shown in Fig 9, the distribution of iterations for all benchmarks indicates that 90% of benchmarks reached the steady state within the first 5 iterations, denoted by the green field. This finding contrasts with the results of the Traini paper, which reported that many benchmarks in their dataset never reached a steady state [9]. Specifically, Traini et al. found that about 11% of benchmark forks never reach a steady state of performance, and 43% of benchmark executions involve at least one fork that doesn't hit the steady state. Our results indicate a significantly higher rate of benchmarks achieving steady-state performance quickly, with only 10% of benchmarks requiring more than 5 iterations to stabilize, represented by the yellow field. This discrepancy suggests potential variability in benchmark behavior, which could be attributed to differences in the datasets or experimental setups used in the two studies. The Traini paper's findings underscore that the assumption of benchmarks always reaching steady-state performance is not always valid and that developers often fail to accurately estimate the end of the warmup phase. In contrast, our study suggests that, under our conditions and configurations, most benchmarks achieve steady-state performance efficiently. These differences highlight the impact of various factors such as benchmarking configurations, the nature of the benchmarks, and possibly the underlying system environments. Our findings show that with optimized configurations and controlled environments, benchmarks can reliably reach a steady state more consistently. This insight is crucial for practitioners aiming to ensure reliable performance evaluations of software systems.

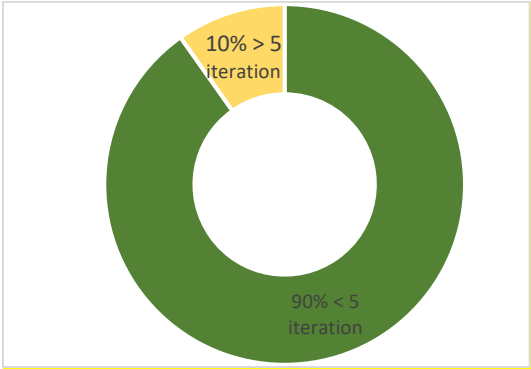

Fig 9: The percentage of iterations for the benchmarking that reaches a steady state.

5.3 Performance Variability

Once the steady state is identified, our focus will shift to measure the remaining variability at this point. Based on our analysis from RQ2 use the identified iteration for each benchmark as the starting point for extracting performance measurement data.

To find the variability of the data from the steady state point to the end of the data, we need to measure the variance and the standard deviation. The variance measures how much the data points deviate from the mean. we calculated as the average of the squared differences from the mean. Where x_i is each data point, and $\bar{x}$ the mean of the data, and n is the number of data points. In our case x_i could be different for each benchmark depending on the steady-state point, which represents the number of iterations required for that specific benchmark. For example, if the first benchmark achieves a steady state at iteration 5, as shown in Fig 5, the variation calculation for this benchmark begins at $(i = 5)$.

$$\text{Variance } s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

In our analysis, we normalized the data to have a mean of 0 and a standard deviation of 1. This normalization process ensures that all features maintain a consistent scale, facilitating statistical analysis. It prevents differences in scale from skewing statistical measures such as means, variances, and correlations. Consequently, our analysis benefits from enhanced interpretability and comparability across different features.

Table 1: Calculation results, with examples of various benchmarks.

Benchmark example	Mean	Range	std	variance
eclipse__eclipse-collections in Fig 10	0.078	0.080	0.006	4.493e-05
apache__camel in Fig 11	3.31e05	8.59e-05	4.98e-06	2.48e-11
jgrapht__jgrapht in Fig 12	0.046	0.255	0.005	3.13e-05
eclipse__rdf4j in Fig 13	0.210	0.078	0.008	6.70e-05

Furthermore, we examined the distribution of data, focusing on the variability captured by standard deviations from the mean. Standard deviations offer a measure of how data points are dispersed around the mean, with the first standard deviation encompassing approximately 68% of the data in a normal distribution and the third standard deviation covering about 99.7% of the data. By considering the third standard deviation, we gain a more comprehensive understanding of the distribution's variability and can identify outliers or extreme values effectively.

In Figure 10, we illustrate the evolution of the data distribution and demonstrate how measures of variability change over time. Specifically, we calculated the standard deviation from the point of steady state to the end of the data. The visual representation in Figure 10 highlights the presence of outliers, which are data points that are outside the range captured by the third standard deviation. It's important to note that the presence of outliers is standard for real-life performance measurements and can provide valuable insights into system behavior.

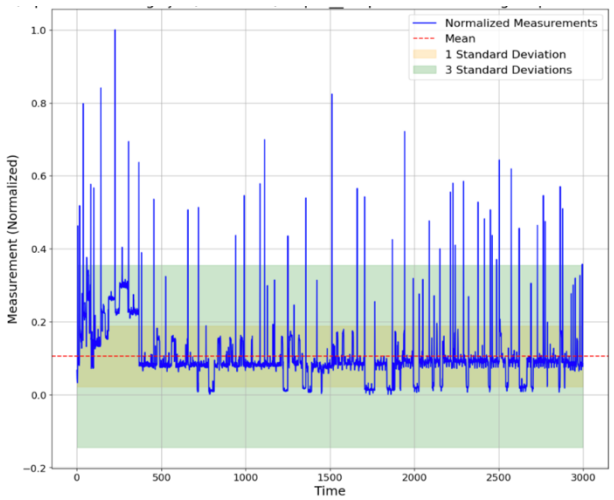

Fig 10: Distribution of Data with Variance for (eclipse__eclipse-collections#org.eclipse.collections.impl.jmh.map.ChainMapPutTest.hppc#isPresized=false&loadFactor=0.75f&size=750000.json)

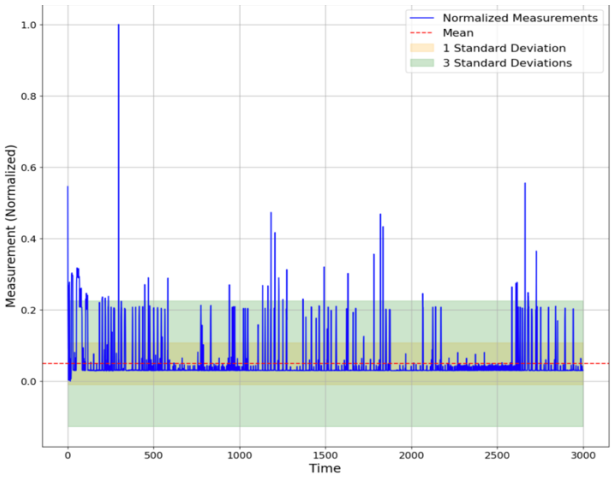

Fig 11: Distribution of Data with Variance for (apache__camel#org.apache.camel.converter.ConverterBenchmarkTest.benchmarkConversionStringToChar#.json)

When comparing the data presented in Fig 10 & 11, we observe that the variance for the benchmarks in Fig 10 is calculated to be 4.493e-05. In this figure, the yellow region represents one standard deviation from the mean. This region extends approximately 0.006 standard deviations below and above the mean, showcasing the typical spread of data around the central tendency. However, despite this, the data points are not closely clustered around the mean, indicating high variability or dispersion within the dataset.

In Fig 11, the standard deviation above and below the mean is approximately 4.98e-06 for the project (apache__camel). Comparing these two figures, it becomes evident that in Fig 10, despite the narrowness of the yellow region, the dataset is spread out over a wider range from the mean, suggesting high variance. Conversely, in Fig 11, with a variance of 2.48e-11, the data points are closely clustered around the mean, indicative of low variance.

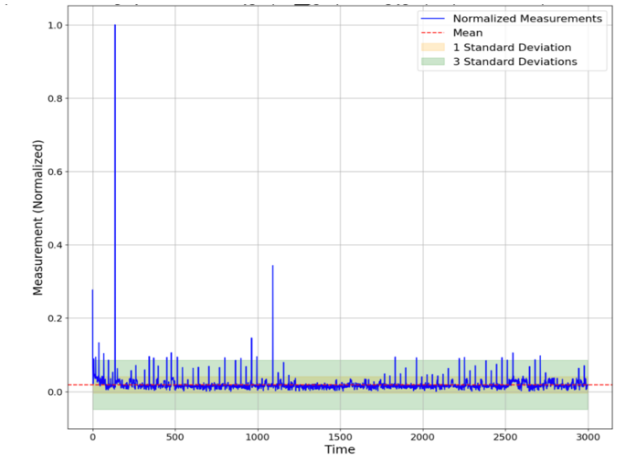

Fig 12: Distribution of Data with Variance for(jgrapht__jgrapht#org.jgrapht.perf.shortestpath.KShortestPathsPerformance.testSuurballeKDisjointShortestPaths#k=50&n=100&numberOfQueries=20&p=0.3.json)

We observed in Fig 12 a small standard deviation which is 0.008 indicating that the data point closer to the mean gives us a tight curve around the mean and the variance after calculation is 3.13e-05.

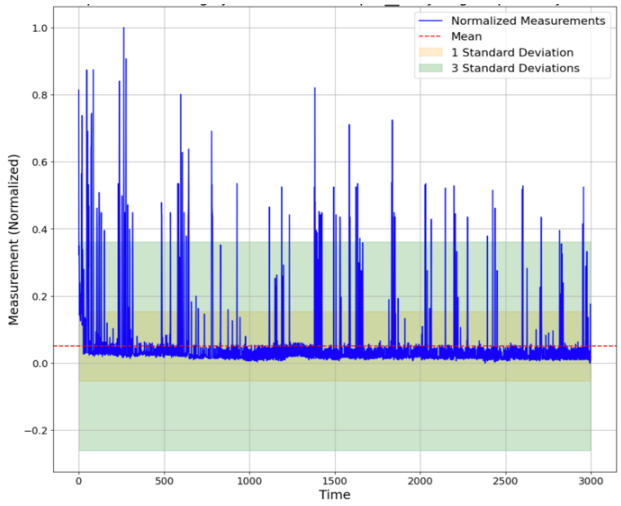

Fig 13: Distribution of Data with Variance for (eclipse__rdf4j#org.eclipse.rdf4j.benchmark.QueryOrderBenchmark.selectDistinct#countk=50&limit=10&syncThreshold=0.json)

In Fig 13 the standard deviation is 0.005 indicating that the data point spread out around the mean resulting in a wider curve compared with Fig 12. The variance 6.70e-05 was much bigger than that of Fig 12, indicating more variability in the data. That suggests that the "jgrapht__ jgrapht" benchmark exhibits lower variability compared with "eclipse__rdf4j". Additionally, Fig. 13 shows more outliers than Fig. 12, which shows fewer outliers. These outliers go beyond the range of three standard deviations from the mean, indicating a higher degree of variability in the data distribution.

By analyzing how the values are distributed throughout the variance's range, we may gain a better understanding of the dataset's characteristics.

Fig 14 shows the distribution of variance values throughout our data, where the y-axis represents the density and x-axis represents the variance for our data measurements.

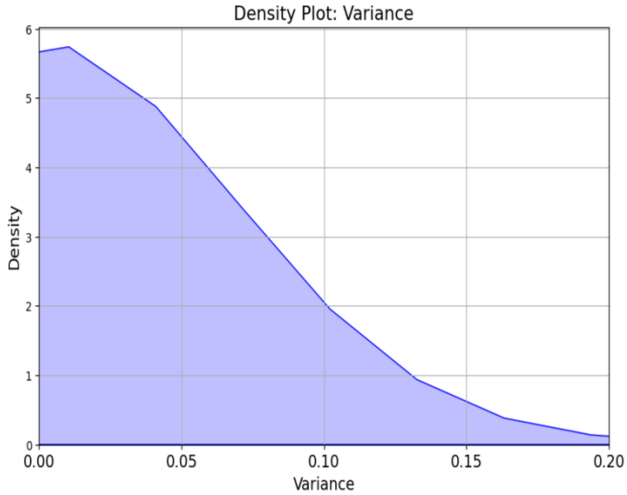

Fig 14: The density plot shows the distribution of variance values across our data

The shape of the density plot can give insights into the distribution of variance in the dataset. The density plot for variance starts at a density of 5.5 when the variance is 0.0 and decreases as the variance increases. We observe that the plot skewed to the left aligns with our observations from the variance analysis in Figures 10, 11, 12, and 13. With fewer data points having a lower variance, a left-skewed density plot suggests that there is a higher concentration of data points at the higher end of the variance shade. This aligns with our findings, particularly in Figures 12 and 13, where we observed smaller standard deviations and tighter curves around the mean, indicative of lower variance. Conversely, in Figures 10 and 11, where the variance was higher, we observed a wider spread of data points away from the mean, contributing to the left skew in the density plot. The density plot's left skewness, which highlights the dataset's variability and the distribution of data points around the mean, supports our interpretation of the variance analysis. High density in the lowest variance measurements (fig. 14), usually in the range of 0.0 and 0.01 indicates that a significant number of the data points in our dataset have very low variability or dispersion from the mean. This high density indicates that a substantial proportion of our data points have variance values clustered around the lower end of the shade. It implies that these data points are relatively consistent or stable in their values, showing little deviation from the mean.

6. Disscusion

The result for RQ1 provides important differences in the average duration of performance testing across numerous open-source projects. The observed trends indicate that the time of performance tests is determined by a wide range of factors, including the specific values and parameters used throughout the testing process. For instance, the "prestodb" project exhibited the highest duration among all projects analyzed, while the "netty" project displayed the lowest duration. This variation can be attributed to differences in the parameters and configurations employed within each project's performance tests. With a thorough evaluation, it was discovered that some benchmark factors contributed to extended test durations.

Parameters such as "blockSize=1000000," "nestedLevel=2," and "numberOfIteration=1000" in the "prestodb" project, for example, were linked to greater duration. These parameters are expected to impact aspects such as data processing size, built data structure complexity, and the number of benchmarking phases, increasing the total test time. Projects with shorter test periods, such as the "netty" project, were distinguished by features that encouraged efficient data handling and processing.
Parameters such as "size=7," "alignedCopyByteBuf=true," and "readonlyByteBuffer=false" are likely to have contributed to quicker test completion times by increasing data processing efficiency.

Furthermore, the frequency distribution of test durations throughout the sample indicated that average duration and frequency are related. This implies that longer test durations are less common, showing rare cases of long testing durations rather than consistent trends across the dataset. These findings highlight the role of parameter settings in determining the efficiency and duration of performance testing. They also emphasize the need for parameter sensitivity analysis and optimization strategies for improving testing processes and achieving consistent performance outcomes in software development projects.

The analysis result of RQ2 focused on determining the stability of benchmark projects by observing their performance Measurements over iterations.
Through manual observation of data distribution plots for each benchmark, stability points were identified, indicating when performance measurements stabilized. The results indicated that many benchmarks reached a stable state within the initial 5 iterations, with only a small percentage requiring more iteration to stabilize. This finding emphasizes the efficiency of the benchmark process in achieving early stability, which is crucial for ensuring reliable performance evaluation of software systems.

The analysis conducted for RQ3 focused on measuring the remaining variability in performance measurements once the steady state was identified. Variance and standard deviation (std) were calculated to quantify the spread of data points around the mean. If we have a small standard deviation the measurements will be close and consistent with low variability. A large standard deviation is a high variability in which measurements are far apart and inconsistent. Then when variability is high the predictability is low.

Normalization of the data ensured consistency across different features, facilitating meaningful statistical analysis. Comparing benchmarks "jgrapht__jgrapht" showed lower variability compared to "eclipse__rdf4j". Outliers of benchmarks, as seen in Fig 9 and 12, can significantly impact the interpretation of data and suggest a greater degree of variability within the dataset. These data points, which fall outside the range of three standard deviations from the mean, often represent unusual or unexpected observations that deviate from the typical pattern of the data. Identifying and understanding outliers is essential as they can influence statistical analyses and decision-making processes.

Finally, the density plot (Fig 14) provides valuable insights into the distribution of variance values across our dataset. The left-skewed distribution indicates a higher concentration of data points towards the higher end of the variance shade, with fewer data points exhibiting lower variance. This observation aligns with our previous variance analyses, highlighting dataset variability and the distribution of data points around the mean. Additionally, the high density in the lowest variance measurements suggests a substantial portion of the dataset with minimal variability, indicating relative consistency in these data points. Overall, the density plot enhances our understanding of the variance characteristics within the dataset, contributing to a comprehensive analysis of the data.

7. Validity Threats

7.1 External validity

It is important to acknowledge that our study examined only specific versions of each benchmark project. While we think the selected versions are representative of the concerns we aimed to address, it is important to note that the findings may not apply generally to other versions. Software performance characteristics can vary dramatically between versions, even when using identical parameter values. As a result, our findings may have limited applicability to different versions of the benchmark projects. Future studies should investigate analyzing numerous versions of benchmark projects to provide a more complete picture of performance patterns across software revisions, hence increasing the external validity of the findings.

7.2 Internal validity

These threats involve factors that may affect the accuracy and reliability of the research findings. One potential issue is measurement uncertainty in the performance data, which could bring errors into the analysis. To ensure the accuracy and reliability of our research findings, we implemented rigorous validation and verification processes throughout the data collection and analysis phases. One critical aspect of this process was data cleaning as we mentioned in section 4.2.2.1, which involved identifying and rectifying flaws, mistakes, or conflicts in the dataset. Specifically, we addressed missing values in the performance measurement data, including test durations and repetitions, which can potentially introduce bias or incompleteness into our analysis. To handle missing values effectively, we employed established techniques such as imputation using the mean for numerical data and the mode for categorical data. By replacing missing values with appropriate estimates derived from the dataset itself, we minimized the potential for bias or incompleteness in our analysis.

Additionally, by normalizing the data, we ensured that all variables were on a comparable scale, making it easier to analyze and interpret the results. This normalization process involved adjusting the data so that the mean (average) fell between 0 and 1. By doing this, we could visualize the data more effectively and draw conclusions based on a standardized framework. This step also helped in identifying outliers, which are data points that significantly deviate from the rest of the dataset. By identifying and addressing outliers, we were able to minimize their potential impact on our analysis, thus improving the accuracy and reliability of our research findings.

7.3 Construct validity

One possible danger to construct validity in our work is that our performance metrics may not completely represent the constructs of interest, such as "benchmark stability." By focusing on the first fork for each benchmark rather than all ten forks, we risk not capturing the full extent of performance variability. This choice limits our ability to generalize findings across multiple executions and may lead to an incomplete representation of the benchmarks' stability. Furthermore, a study conducted by a trainee suggested that some benchmarks did not reach a steady state [9]. This discrepancy may be due to our research only analyzing the first fork. Analyzing all ten forks could provide a more comprehensive understanding of whether and when benchmarks reach a steady state. Addressing this limitation would enhance the robustness of our findings and improve our ability to generalize the results across different benchmark executions.

8. Conclusion And Future Work

This paper presents an analysis of performance measurements across 30 Java open-source projects that revealed significant variability in average test durations, indicating diverse testing conditions and parameter settings across projects. The observed differences in duration underscore the importance of parameter configurations in

influencing testing efficiency and duration. Moreover, our investigation into the stability of benchmark projects highlighted the efficiency of the benchmarking process, with many benchmarks reaching a stable state within the initial five iterations. This early stability is crucial for ensuring reliable performance evaluations of software systems.

Furthermore, our examination of variability in performance measurements, quantified through variance and standard deviation calculations, emphasized the impact of parameter settings on data consistency and predictability. The identification of outliers within the dataset underscored the need for robust outlier detection mechanisms to ensure the integrity of statistical analyses.

The density plot revealed that a considerable portion of the dataset exhibited low variability, with data points clustered around the lower end of the variance shade. This suggests that many benchmarks maintained consistent and stable performance measurements, showing little deviation from the mean. In general, the density plot complemented the statistical analysis by visually representing the dataset's distribution and variability, providing valuable insights into the performance characteristics of the benchmarks. Our study sheds light on crucial considerations for both practitioners and researchers in the domain of software performance evaluation.

On the practitioners' side, by allowing users to configure benchmarking parameters and settings, benchmarking systems can accommodate diverse use cases and scenarios encountered in real-world software environments. This practical empowerment enables practitioners to tailor performance tests to their unique applications, infrastructure setups, and performance goals, thereby enhancing the relevance and applicability of benchmarking results.

On the researchers' side, building on the insights gained from our investigation, researchers are encouraged to explore the potential of dynamic reconfiguration techniques to enhance the stability and efficiency of benchmarking processes. Our study suggests that there is significant room for improvement in optimizing parameter configurations and testing conditions to achieve more consistent and predictable performance outcomes. Therefore, future research endeavors could focus on designing innovative methodologies and tools for dynamically adapting benchmarking parameters based on real-time performance feedback. Such advancements hold the promise of revolutionizing the field of performance evaluation and empowering researchers with more robust and reliable performance assessment frameworks.

REFERENCES

[1] 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, 2017, pp. 373-384.

[2] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 119-130.

[3] D. A. Mey et al., "Score-P: A unified performance measurement system for petascale applications," in Competence in High-Performance Computing 2010: Proceedings of an International Conference on Competence in High-Performance Computing, June 2010, Schloss Schwetzingen, Germany. Springer Berlin Heidelberg, 2012, pp. 85-97.

[4] Y. Liu et al., "A reliable open-source package for performance evaluation of floating renewable energy systems in coastal and offshore regions," Energy Conversion and Management, vol. 174, pp. 516-536, 2018.

[5] A. Lazidis, K. Tsakos, and E. G. Petrakis, "Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems," Internet of Things, vol. 19, p. 100538, 2022.

[6] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in Proceedings of the first workshop

[7] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "Performance assessment of IEEE 802.11 p with an open source SDR-based prototype," IEEE Transactions on Mobile Computing, vol. 17, no. 5, pp. 1162-1175, 2017.

[8] D. C. Schmidt and A. Porter, "Leveraging open-source communities to improve the quality & performance of open-source software," in Proceedings of the 1st Workshop on Open Source Software Engineering, May 2001, vol. 1.

[9] Traini, L., Cortellessa, V., Di Pompeo, D., & Tucci, M. (2023). "Towards an effective assessment of steady state performance in Java software: Are we there yet?" Empirical Software Engineering, 28(13). [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x.

[10] H. Taherdoost, "Different Types of Data Analysis; Data Analysis Methods and Techniques in Research Projects," *International Journal of Academic Research in Management (IJARM)*, vol. 9, no. 1, pp. 1-9, 2020.[Online]. Available: https://hal.science/hal-03741837/document

[11] R. Barrett, R. Barrett, R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "Virtual machine warmup blows hot and cold," in Proceedings of the First Workshop on Performance Evaluation of Open Source Software, 2017, pp. 1-5.

[12] Python Software Foundation, "Python: Python Programming Language." [Online]. Available: https://www.python.org/

[13] [13] R. Rahul, "Benchmark Java Applications using JMH," 20-Dec-2019. [Online]. Available: https://dzone.com/articles/benchmark-java-applications-using-jmh. [Accessed: 27-May-2024].