Empirical Software Engineering (2025) 30:157
https://doi.org/10.1007/s10664-025-10712-3

Is code coverage of performance tests related to source code features? An empirical study on open-source Java systems

Muhammad Imran[1] · **Vittorio Cortellessa**[1] · **Davide Di Ruscio**[1] · **Riccardo Rubei**[1] · **Luca Traini**[1]

Accepted: 28 July 2025 / Published online: 30 August 2025

Abstract
Performance testing aims to ensure the operational efficiency of software systems. However, many factors influencing the efficacy and adoption of performance tests in practice are not yet fully understood. For instance, while code coverage is widely regarded as a key quality metric for evaluating the efficacy of functional testing suites, there is limited knowledge about the types and levels of coverage that performance tests specifically achieve. Another important factor, often perceived as a barrier to the broader adoption of performance tests yet remaining relatively unexplored, is their extended execution time. In this paper, we examine (i) the coverage of performance testing suites, (ii) the characteristics of source code associated with performance-tested components, and (iii) the time cost of executing performance tests. Our analysis on open-source Java systems reveals that performance tests achieve significantly lower code coverage than functional tests, as expected, and it highlights a significant trade-off between coverage and execution time. Our results also indicate a lack of generalizable characteristics in the source code covered by performance tests.

Keywords Performance Testing · Unit Testing · JMH · JUnit · Java

Communicated by: Tingting Yu.

✉ Muhammad Imran
muhammad.imran@graduate.univaq.it

Vittorio Cortellessa
vittorio.cortellessa@univaq.it

Davide Di Ruscio
davide.diruscio@univaq.it

Riccardo Rubei
riccardo.rubei@univaq.it

Luca Traini
luca.traini@univaq.it

[1] University of L'Aquila, L'Aquila, Italy

1 Introduction

Software performance is a critical non-functional aspect of software systems. Deterioration in performance can present significant business challenges, including user dissatisfaction (Brutlag 2009) and financial losses (Olenski 2016). To address these concerns, organizations typically employ performance testing (Weyuker and Vokalos 2000), a technique designed to evaluate the software system's performance before its deployment in a production environment. However, the practical implementation of performance testing faces challenges.

One of such challenges is associated with the development and maintainability of performance testing suites. Creating these tests often requires specific technical expertise that may not be readily available to the average developer (Behutiye et al. 2020; Traini 2022). Additionally, software development processes tend to prioritize functional development activities, which can lead to limited resource allocation for quality assurance tasks (Alsaqaf et al. 2019; Behutiye et al. 2020), such as the creation and maintenance of performance tests (Traini 2022). These factors collectively contribute to the oversight of performance assurance activities, leading to potential implications on the *quality* of performance testing suites.

The traditional way of assessing the *quality* of a testing suite involves using *code coverage*. This metric gauges the extent to which the testing suite executes the software source code, serving as a simple yet effective indicator of the testing suite quality. Although the validity of code coverage as a measure of test quality is still a matter of debate (Inozemtseva et al. 2014; Gopinath et al. 201), it remains the de-facto standard for evaluating testing suites in practical scenarios. Code coverage has traditionally been used and studied in the context of functional testing (Ivanković et al. 2019; Piwowarski et al. 1993; Inozemtseva et al. 2014; Gopinath et al. 201), such as unit tests. However, there is growing evidence that its relevance extends to software performance testing as well. For instance, Ding et al. (2020) demonstrated that test coverage has a significant impact on the ability of performance tests to detect performance bugs. Similarly, Jangali et al. (2023) introduced a technique to automatically generate performance tests from existing functional tests, finding that higher coverage leads to better detection of performance issues. Compared to real-world performance testing suites, this approach resulted in notably better performance test coverage and an enhanced ability to uncover performance issues. Moreover, code coverage has proven useful in various automated techniques designed to enhance performance testing. These include methods for performance test selection and prioritization (Reichelt and Kühne 2018; Laaber et al. 2021; Grambow et al. 2022), as well as strategies for generating performance-relevant inputs (Lemieux et al. 2018; Noller et al. 2018).

Despite these indications, there is still limited knowledge about the code coverage achieved by performance tests. In a previous conference paper (Imran et al. 2024), we took an initial step on this issue by conducting an investigation into the code coverage of performance tests and the associated execution time cost, which is an important factor typically in trade-offs with code coverage (Yoo et al. 2012). Our methodology involved selecting 28 Java software systems on GitHub that featured JMH benchmarks, a widely used form of performance tests in Java. We then conducted a comprehensive analysis to extract all Java methods within these software systems. Finally, we executed 2,190 JMH benchmarks on these Java software systems to identify the methods covered by performance testing. Our findings revealed that JMH benchmarks achieve a limited code coverage of about 8.8% on average, which is only a quarter of that achieved by the JUnit tests. Additionally, JMH

benchmarks incur a considerably high execution time cost, which is in average 62 times larger than those of JUnit tests.

This paper extends our previous one (Imran et al. 2024), with the main intent to investigate whether static features can play the role of "detectors" of code that is usually covered by performance tests. The identification of one or more such features, in perspective, would support the work of performance testers by pointing them to portions of code that claim their attention, only on the basis of static code analysis, therefore without needing to execute those portions of code. With this main motivation in mind, we introduced a wider analysis that explores the correlation between the characteristics of code components and their likelihood of being covered by performance tests. Specifically, we aim to understand whether code components covered by performance tests share characteristics (e.g., recurrent patterns) that are generalizable across software systems. The identification of such patterns can help, in particular: (i) to guide developers in selecting which code components to test, and (ii) to improve performance engineering techniques, such as performance test prioritization (Laaber et al. 2024, 2021), selection (De Oliveira et al. 2017), and generation (Jangali et al. 2023; Rodriguez-Cancio et al. 2016).

In order to investigate this aspect, we extracted 44 well-known source code metrics from 177, 697 Java methods, and we trained machine learning classifiers to predict whether a method is covered by a performance test based on these metrics. Through this process, we searched for correlations between the code metrics and the likelihood of being covered by performance tests. We employed a rigorous methodology that involved training 11 machine learning algorithms across 99 configurations, while adhering to best practices such as dataset rebalancing, feature selection, and hyper-parameter tuning. Our results basically exclude correlations between source code metrics and performance test coverage, by suggesting a lack of generalizable recurrent patterns in performance-tested methods. This finding indicates that the choice of code components to be performance tested is primarily driven by domain-specific objectives rather than generalizable code characteristics. Consequently, this poses significant challenges for the automated generation, selection, and prioritization of performance tests.

To sum up, this paper provides the following contributions:

- An empirical investigation on the code coverage of performance testing, and the associated execution time cost.
- An extensive investigation on the correlation between source code metrics and performance testing coverage.
- A replication package (Imran et al. 2024) containing the dataset we mined in the study, the scripts we used to perform the data analysis, and the detailed results of our analysis.

2 Background

2.1 Performance Testing

Performance indicates how well a software system or component operates in terms of efficiency, including aspects like response time and throughput. The *response time* refers to the time required to respond to a request, while the *throughput* of a system is the number of

requests that can be processed in some specified time interval (Smith and Williams 2002). Just like functional correctness, performance is one of the key characteristics in different classifications of the software quality attributes (Chung et al. 2012; ISO/IEC n.d.).

A common practice to determine the performance of any application is performance testing Zhao et al. (2019). Performance testing is an activity aimed at evaluating the efficiency of a software system in a pre-production environment, often in terms of response time, throughput, or a combination of both (Vokolos and Weyuker 1998). It includes a wide variety of approaches, such as system-level testing and load testing (Weyuker and Vokalos 2000). One of the goals of performance testing is to identify problems that cause the system performance to degrade. Even if a software system lacks explicit performance specifications, it is implicitly required that it should not take infinite time or resources to execute.

Performance testing is usually carried out by repetitive executions of target code until enough performance data points are collected (He et al. 2019a, 2021). To evaluate overall software systems, state-of-the-art performance testing practices typically involve relatively large-scale and long-running tests (Jiang and Hassan 2015). However, this process should be systematic due to the expensive nature of performance tests in terms of execution time cost (Imran et al. 2024). To deal with this intrinsic complexity, several approaches to performance unit testing, such as microbenchmarking, have been proposed for accurate performance evaluation of small-scale and isolated source code segments (Jangali et al. 2023).

Microbenchmarks are small test cases used to evaluate the performance of a specific piece of code or function (Eadline 2005). They are increasingly becoming popular and widely adopted, mainly due to their short runtimes and testing durations (Laaber and Leitner 2018). Performance microbenchmarking aims to test smaller code units (e.g. methods or statements) (Shipilev 2014; Costa et al. 2019). This approach is particularly important in languages like Java, where system performance can be significantly affected by apparently small code segments. Microbenchmarking is used to evaluate specific performance metrics such as execution time, memory consumption, and throughput, while providing detailed insights on the performance characteristics of individual code units.

However, there are inherent challenges to performance microbenchmarking, like unreliable results (Kalibera and Jones 2020), and a lack of appropriate tools (Stefan et al. 2017). Due to these challenges, many frameworks provide benchmark tooling like Caliper (Google 2021), AutoJMH (Rodriguez-Cancio et al. 2016), and JUnitPerf (O'Connor n.d.). For Java-based software, JMH (Corporation 2016) is the de-facto standard framework used to automate performance microbenchmarking by defining and executing software benchmarks. Therefore, in this study, we focus on microbenchmarking using JMH in Java programs where we consider methods as units to be tested.

2.2 Test Coverage

Test coverage is a commonly used metric in software testing that calculates how thoroughly the test cases exercise a given program. It is used as an indicator to monitor the quality of a test suite and a way to measure how thoroughly the software is tested. Test coverage refers to the portion of a software application used during a particular test execution (Alves and Visser 2009). Test coverage also serves to inform analysis techniques, such as fault localization, or to guide search-based algorithms towards generating coverage-optimized test suites (e.g., Rohella and Takada 2018; Zhou et al. 2022).

Typically, test coverage criteria include statement coverage and branch coverage at the level of methods, classes, packages, and overall system. Statement coverage measures the percentage of executable statements exercised by a test suite, while branch coverage measures the percentage of branches exercised by a test suite. In this paper, the coverage granularity is considered at the method level. This choice stems from technical conflicts between traditional statement-level coverage tools and JMH microbenchmarks. We provide more details on this aspect in Section 6.

The coverage can be determined using various test coverage tools. The coverage measurement is performed by inserting measurement probes into a program at the targeted granularity level, such as at a statement, block, or method level, and then running the tests on the instrumented program, while typically storing coverage information in trace files (Häubl et al. 2013). All the tools usually work on this principle but they may vary in the languages to which they apply. Also, they may provide measurements at different granularity levels such as statement, branch, method, and class (Yang et al. 2006).

Despite limited applications and attention to test coverage for performance testing, better test coverage has proven to be effective in finding synchronization performance bugs in production environments (Alam et al. 2017). Applying test case prioritization, coverage-based techniques provide significant benefits in regression testing by efficiently uncovering performance regressions (Laaber et al. 2021). Studies have shown that dynamic coverage analysis (e.g., line coverage or branch coverage that involve measuring the test coverage while the program is running) can effectively measure the execution behavior of Java programs. These techniques prove to be effective in early capturing significant performance changes (Brown et al. 2005). Due to this, coverage-based approaches are particularly beneficial for performance testing of software microbenchmarks, which typically require longer execution times than unit tests (Imran et al. 2024).

3 Study Design

This study investigates the code coverage achieved by *performance tests* and their time cost, given the relevance of this aspect for the practical usage of performance tests. To aid the interpretation of our results, we conduct a comparative analysis with *functional tests* to assess the differences both in terms of code coverage and time cost. Furthermore, we explore the potential relationships between source code metrics and the likelihood of the analyzed source code being covered by performance tests. By identifying these relationships, we aim to uncover recurring patterns in the source code that can inform and guide performance testing efforts.

We focus on JMH microbenchmarks and JUnit tests due to their extensive use within the Java ecosystem. JMH is the de-facto standard for developing and running microbenchmarks, a widely known form of performance tests in Java software (Leitner and Bezemer 2017), while JUnit stands as one of the most popular libraries for implementing and executing Java functional tests. In this study, we aim to address the following research questions (RQs):

▷ RQ_1: *To what extent do performance tests cover the source code of software systems?* Our objective is to evaluate the extent of code coverage achieved by performance testing suites across various software systems and compare it to the coverage provided by func-

tional testing suites. We assess coverage from multiple perspectives, including overall and direct coverage, and we analyze the degree of overlap among different tests in their coverage of identical sections of the source code.

▷ RQ_2: *What is the time cost of performance tests?* We assess the time costs incurred during the execution of performance tests, and compare them to those associated with functional tests. We measure the overall time consumed at *suite-level* and *test-level*, thus providing insights on the temporal impact of performance testing at different granularity.

▷ RQ_3: *Are there any specific characteristics in the source code that can guide performance testing?* To determine if there are any peculiarities in the source code sections chosen for performance testing, we investigate the relationship between static code features and the likelihood of being covered by performance tests. To do so, we leverage machine learning models to predict whether a method is performance tested or not. Specifically, we employ a carefully designed process consisting of different steps, while adhering to current machine learning best practices largely inspired by a similar work (Laaber and Leitner 2018). We start with machine learning models using default settings. Then, we apply both class rebalancing techniques (Chawla et al. 2002; Batista et al. 2004) and feature selections (Guyon et al. 2002; Jiarpakdee et al. 2018). Finally, we adopt hyperparameter tuning (Avinash et al. 2022) to enhance the prediction accuracy of the models. High prediction accuracy indicates the presence of distinctive patterns in the source code that are subject to performance testing. Conversely, low prediction accuracy suggests a lack of generalized characteristics in performance tested methods. To address RQ_3, we investigate the following three sub-research questions:

- $RQ_{3.A}$: *Can we predict whether a method should be performance tested using machine learning models with default settings?* We investigate the use of machine learning models with their default settings to evaluate if static code features can be used to predict whether a code component should be covered by performance tests.
- $RQ_{3.B}$: *How does the prediction accuracy change when employing feature selection and class-rebalancing techniques?* We employ two well-known preprocessing steps, namely *(i)* feature selection and *(ii)* class-rebalancing, to investigate their impact on the prediction accuracy of machine learning models. For feature selection, we remove less relevant code features, such as co-linear and multi-co-linear features, from our dataset. Furthermore, we conduct an analysis of the features' importance by calculating SHAP (Shapley Additive exPlanations) values.[1] For class-rebalancing, we apply both oversampling and undersampling techniques.
- $RQ_{3.C}$: *How does the prediction accuracy change after tuning the hyperparameters of machine learning models?* We aim to investigate the extent to which the optimization of hyperparameter configurations can enhance the prediction accuracy of machine learning models.
- $RQ_{3.D}$: *Can transformer-based models, such as CodeBERT and GraphCodeBERT, improve the prediction accuracy compared to traditional machine learning models?* To assess the potential of deep learning approaches for this task, we explore the use of transformer-based models that have been pre-trained on code (i.e., CodeBERT and GraphCodeBERT). These models are capable of learning rich semantic representations of source code without requiring manual feature engineering. We investigate whether

[1] https://shap.readthedocs.io/en/latest/

such models can better capture complex patterns that can guide performance testing decisions.

3.1 Main Steps of the Performed Study

Given the objective of our study, we selected an initial pool of 40 Java software projects hosted on GitHub. This selection was guided by four key considerations: *(i)* these systems are well-established Java libraries that cover a broad spectrum of application domains, *(ii)* each of these projects includes JMH benchmarks and JUnit tests, *(iii)* we are familiar with the commands necessary to execute the JMH microbenchmarks for these systems, and *(iv)* they have been used in prior work (Laaber and Leitner 2018; Laaber et al. 2021, 2020; Traini et al. 2021, 2022), thus supporting their appropriateness in this context.

As shown in Fig. 1, the executed process consists of three main steps: *(i) raw data collection* from the selected software systems, *(ii) data wrangling*, and *(iii) model training and tuning* phase, as described in the following subsections.

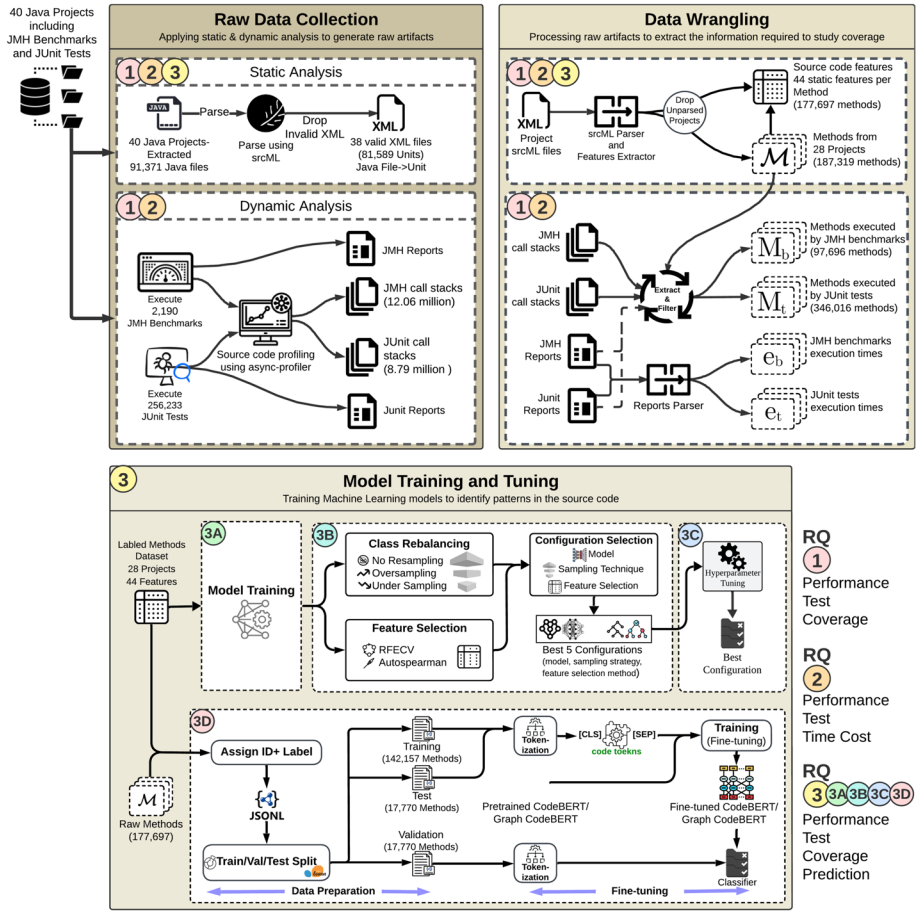

Fig. 1 Main steps of the performed study

3.1.1 Raw Data Collection

Our data collection combines both static code analysis and dynamic analysis of test executions. In particular, we collected three distinct types of raw data from each Java system, i.e., *srcML XML files*, *JMH/JUnit callstacks*, and *JMH/JUnit reports*.

▷ *srcML XML files:* We transformed each Java source file into a structured XML format using srcML toolkit (Collard et al. 2011). This transformation facilitates a structured and reliable static analysis of the source code, thus enabling straightforward extraction of relevant code information, such as the list of Java method signatures and other source code elements corresponding to Java language features for specifying and managing control flows, data, and concurrency that appear in the project. In this process, we encountered a specific issue with two projects, namely *apache hive* and *eclipse jersey*, where the srcML toolkit generated incomplete XML files that omitted the representation of specific Java source files. Thus, we decided to exclude these two projects from our analysis. As a result, we successfully parsed the source code of 38 of our initially selected projects.

▷ *JMH/JUnit callstacks:* We rely on dynamic analysis to identify the code components covered (or not covered) by JMH/JUnit tests. We prefer dynamic analysis over static analysis due to several limitations of the latter (Walker et al. 2020), such as its inability to reliably derive method invocations. Specifically, we employed *async-profiler*[2] to profile the execution of JMH benchmarks and JUnit tests by capturing their respective call stacks. A call stack reports the currently active methods in the CPU and the sequence of their invocations. For illustration, consider Fig. 2, which presents a call stack from the execution of a JMH benchmark named `skinnyEncodeIntoCompressedByteBuffer`. Through this call stack, we can deduce the sequence of executed methods within the benchmark. From Fig. 2, we can observe that the benchmark first calls `encodeIntoCompressedByteBuffer`, which subsequently invokes `encodeIntoByteBuffer`, and this is followed by `writeCountsDiffs`, and so forth.

After executing each testing suite, we produced a distinct file that catalogs all unique call stacks observed during the execution. However, during this procedure, we faced issues attaching the profiler to the JMH benchmarks of seven projects. Consequently, we excluded these projects from our analysis. We faced similar issues for JUnit testing in nine projects. As a result, we collected 12.06 million unique call stacks for JMH benchmarks from 31 systems, and 8.79 million unique call stacks for JUnit tests from 19 projects.

▷ *JMH/JUnit reports:* We collected JMH and JUnit reports to obtain two primary information: *(i)* the list of JMH and JUnit tests and *(ii)* their corresponding execution times. For the latter metrics, we have exploited JMH configuration mechanisms, which allow developers to declare the number of repetitions for each JMH test, thus deducing a bound on the time needed to execute the JMH test.

We retained the JMH configurations defined by the developers (e.g., number of forks, warmup iterations, and measurement iterations) to reflect the intended benchmarking setup in each project. This ensures that our analysis captures real-world performance testing practices.

[2] https://github.com/async-profiler/async-profiler

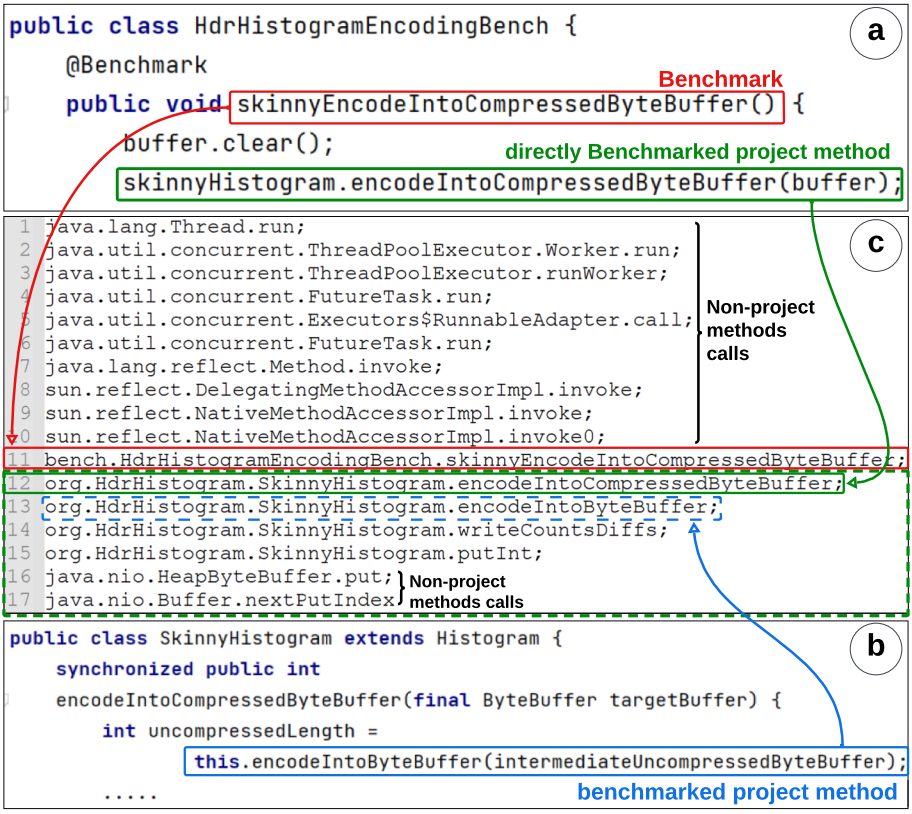

Fig. 2 (a) Example invocation of a method from a benchmark. (b) Benchmarked method definition and indirect call to another project method. (c) Example of a collected call stack

To obtain the JMH configurations for each JMH test, we applied an approach similar to a prior work (Traini et al. 2022), which exploits a JMH feature that allows to overwrite configurations on the fly via CLI arguments. We executed each JMH test while reducing the execution time through JMH CLI arguments,[3] and we stored the associated JMH reports, which include the JMH configurations set by developers. To gather execution times of JUnit tests, we utilized the Maven Surefire plugin.[4] This plugin is an established instrument within the Java ecosystem, and it is explicitly tailored for running JUnit tests through Maven. The execution of tests produces XML reports that break down the execution time for each JUnit test.

We conducted all tests on a dedicated machine equipped with Linux Ubuntu 18.04.2 LTS, powered by a dual Intel Xeon CPU E5-2650 v3 at 2.30 GHz, boasting 40 cores and 80 GB of RAM.

[3] We refer the reader to Traini et al. (2022) for a detailed explanation of this process.

[4] https://maven.apache.org/surefire/maven-surefire-plugin/

3.1.2 Data Wrangling

To address our research questions, we focus on four key pieces of information for each Java system: *(i)* the entire set $\mathcal{M}$ of Java methods appearing in source code, *(ii)* the set of features for each method (as listed in Table 1), *(iii)* the set M_t of methods covered by each (JMH/JUnit) test *t*, and *(iv)* the execution time e_t of each (JMH/JUnit) test *t*. In the following, we describe the process used to derive this information, starting from the raw data.

▷ *Java methods:* To extract the fully qualified names of all methods within each project, we parsed the *srcml XML files* using the *lxml* library and employing XPath queries. While

Table 1 Collected source code features

Category/Level	Feature Name	Description
(i) meta information		
Class	classScope	access specifier of the class
	cIsAbstract	if the class is declared abstract
	cNestingLevel	nesting depth of the class (0 for no nesting)
	#cImports	no. of import statements
	#cNativeImports	no. of native imports
	#cMethods	no. of methods
	#cInherits	if the class uses inheritance
	#cImplements	no. of interfaces implemented
	#classLOC	Lines of code of the class
	#cPkgClasses	no. of classes in current Package
Method	#methodLOC	Lines of code of the method
	#nameLen	method name length
	methodScope	access specifier of the method
	isOverloaded	if the method is overloaded
(ii) language feature		
Control flow and data	#if	no. of if conditions
	#switch	no. of switch statements
	#case	no. of case statements
	#for	no. of for loops
	#while	no. of while loops
	#do	no. of doWhile loops
	#nestedLoops	no. of nested loops (arbitrary depth)
	#methodCalls	no. of methods called
	#internalCalls	no. of internal methods called
	#externalCalls	no. of external methods called
	#return	no. of return statements
	#throw	no. of throw statements
	#catch	no. of catch statements
	#cyclo	cyclomatic complexity by McCabe (1976)
	#vars	no. of the variables declared
(iii) calls to standard library APIs		calls to methods from selected standard libraries in Java (details are given in Table 2)

executing XPath queries on XML files, facilitated by *lxml*,[5] we encountered a limitation regarding the size capacity. Specifically, there was a size threshold (i.e., 6,800 srcML units) over which *lxml* could not evaluate the given XPath expressions. This constraint necessitated the exclusion of three projects from our analysis. Ultimately, this process resulted in extracting a set $\mathcal{M}$ of Java methods for each Java system. In total, we extracted 187,319 methods from 28 distinct systems (as reported in Table 5, where detailed information about the amount of *Methods*, *Benchmarks* and *Unit Tests* per project is provided).

▷ *Source code features:* To extract the features from the source code, we employed additional XPath queries on the *srcml XML files*. Specifically, for each Java method in $\mathcal{M}$, we extracted a set $\mathcal{F}$ of 44 source code features. The complete list of considered features is available in Table 1. To maintain consistency, we have adhered to specific naming conventions for the employed features. In particular, all features are named using Camel case notation. Moreover, quantitative feature names begin with the prefix "#", whereas features at the *class granularity level* include the prefix "c".

We considered three kinds of source code features: *(i) meta information*, e.g., the number of LOCs or files (these features were collected at both class and method granularity levels); *(ii) language features*, e.g., loops, conditionals, or variables; and *(iii) calls to standard library APIs*. We have chosen these kinds of source features because of their relationship with software performance, which might influence the likelihood of the Java method being performance tested or not (Chen et al. 2022). Previous research on software performance often leverages statically or dynamically determined source code features, such as added loops or method calls, to predict whether a code change slows down (or speeds up) the program under analysis (Alcocer et al. 2020).

Below, we provide a more detailed description of each kind of source code features.

(i) **Meta Information:** Meta information features provide insights into the structure and characteristics of the code, which can potentially impact the software performance. We collected these features at both the class and method granularity levels. At the class level, features such as the number of import statements (#cImports), number of methods (#cMethods), and number of lines of code in a class (#classLOC) offer a high-level view of the codebase. At the method level, features such as the number of lines of code in a method (#methodLOC), the method name length (#nameLen), and the access specifier of the method (methodScope) provide more fine-grained information. Research has shown that meta information features, including file and method granularity features, have been successfully employed in prediction models for performance properties (Chen et al. 2022).

(ii) **Language Features:** They are related to control flow and data elements, which have been extensively used in prior work (Laaber et al. 2021; Huang et al. 2014; Chen et al. 2022). Control flow elements include language constructs to specify and manage conditions (#if, #switch, #case), loops (#for, #while, #do, #nestedLoops), function lifecycle (#methodCalls, #internalCalls, #externalCalls, #return), exception handling (#throw, #catch), and cyclomatic complexity (#cyclo). These elements contribute to the increased complexity of a function, which may negatively impact software performance. Research has frequently identified loops as a root cause of performance issues (Jin et al. 2012a; Alcocer and Bergel 2015; Selakovic and Pradel 2016; Zhao et al. 2020), and control flow elements have been employed in machine-learning-based

[5] https://lxml.de/

performance prediction models (Chen et al. 2022; Ding et al. 2020). Also, the method memory usage and stack size may potentially affect performance due to increased garbage collection (GC) activity. So, more variables (#vars) increase pressure on the GC, thus leading to more frequent memory allocations and deallocations, which can impact performance (Costa et al. 2017).

(iii) **Library Calls:** Standard library packages provide functionalities such as file and network I/O, communication with the operating system (OS), text and string processing, and concurrency primitives. These functionalities often involve non-deterministic behaviors, such as waiting for locks, blocking during I/O operations, and network data transmission, which can affect software performance. Consequently, calls to certain standard libraries may adversely impact software performance. Additionally, inefficient or incorrect use of APIs, as well as concurrency and synchronization issues, have been identified as common sources of performance bugs (Jin et al. 2012a; Alcocer and Bergel 2015; Selakovic and Pradel 2016; Zhao et al. 2020). To analyze standard library calls, we group them as single features at the package level. All calls to a specific package are combined into one feature. Table 2 shows the Java standard packages mapped as

Table 2 Standard library call features from Table 1

Package	Feature Name	Category	Description
java.util	usesJavaUtil	utility	Utility classes in Java.
java.lang	usesJavaLangThread	concurrency	Classes for multi-threading and concurrent programming.
java.util.concurrent	usesJavaUtilConcurrent	concurrency	Advanced concurrency utilities & collections.
java.io	usesJavaIo	io	Classes for I/O through data streams.
java.nio	usesJavaNio	io	New I/O for more scalable I/O operations.
java.nio.channels	usesJavaNioChannels	io	Channels and selectors for non-blocking I/O.
java.nio.file	usesJavaNioFile	io	File I/O enhancements using the NIO framework.
java.nio.charset	usesJavaNioCharset	io	Charset classes for encoding and decoding.
java.net	usesJavaNet	io	Networking classes for implementing protocols and communication.
javax.net.ssl	usesJavaxNetSsl	io	SSL/TLS support for secure network communication.
java.lang	usesJavaLang	os	Core Java language classes and basic types.
java.lang.management	usesJavaLangManagement	os	Management interfaces for the Java platform.
java.util.regex	usesJavaUtilRegex	strings	Regular expressions for pattern matching and string manipulation.
java.text	usesJavaText	strings	Classes for parsing and formatting text.
java.math	usesJavaMath	math	Mathematical functions and utilities.

boolean features with their descriptions.▷ *Test coverage:* We leveraged the JMH and JUnit call stacks to identify the Java methods covered by each JMH benchmark or JUnit test. For each test *t*, we extracted the set M_t of project methods executed within *t*. To achieve this, we iterated over all the project methods in $\mathcal{M}$ and checked if each method appeared after *t* in at least one call stack. A method *m* is considered covered by test *t* if it is invoked either directly or indirectly by *t* (i.e., $m \in M_t$). Additionally, for each test *t*, we created a separate set $\hat{M}_t$ that only contains the methods directly called by *t*. In other words, for each test *t*, we select the methods in $\mathcal{M}$ that appear immediately after *t* in the call stacks. For instance, in the example shown in Fig. 2, the method directly invoked by the benchmark skinnyEncodeIntoCompressedByteBuffer would be encodeIntoCompressedByteBuffer, but not encodeIntoByteBuffer. At the end of this process, for each JMH benchmark or JUnit test *t*, we obtain two sets: M_t representing methods covered either directly or indirectly by *t*, and $\hat{M}_t$ representing methods directly covered by *t*.

▷ *Test execution time:* The JMH configuration set by developers determines the execution time of a JMH benchmark. This configuration defines the levels of repetitions (i.e., forks, iterations, and invocations) used during benchmarking to address the inherent variability of performance measurements (Kalibera and Jones 2013). Invocations are repeated benchmark executions within a time-bound iteration, while a series of iterations forms a fork. Each fork usually comprises two distinct types of iterations: warmup and measurement iterations. Warmup iterations are intended to bring the fork into a steady state of performance (Traini et al. 2022; Kalibera and Jones 2013), while measurement iterations are the ones that are actually used for performance assessment. For each benchmark *t*, we first extract the JMH configuration from the JMH reports, i.e., the warmup iteration time *w*, the measurement iteration time *r*, the number of warmup iterations *wi*, the number of measurement iterations *i*, and the number of forks *f*. Then, we compute the associated execution time e_t accordingly: $e_t = (w \cdot wi \ + \ r \cdot i) \cdot f$.

For JUnit tests, we instead directly extract the execution time e_t for each test *t* from the Surefire XML reports.

3.1.3 Model Training and Tuning

We use machine learning models to identify recurring patterns in source code features that correlate with performance testing coverage. By employing classification algorithms, we aim to predict the suitability of a method for performance testing based on these features. We draw inspiration from the study conducted by Laaber et al., (2021) where the authors applied machine learning techniques to classify stable and unstable Go benchmarks. Their study analyzed 230 open-source Go projects, encompassing a total of 4,461 unique benchmarks. Similar to our work, they faced the challenge of working with an unbalanced dataset. The process applied in this paper involves several steps: training the classifiers with a dataset that includes source code features of methods and coverage information as labels to indicate whether a method is performance tested. We also explore alternative class rebalancing techniques, utilize feature selection methods, and perform hyperparameter tuning to optimize the prediction accuracy of our models. Specifically, our process involves three progressively improving steps, as detailed below by referring to Fig. 1. First, we train the machine learning models using their default settings ($RQ_{3.A}$). Next,

Table 3 Employed ML models

Model	Acronym	Algorithm
Random Forest	RF	Ensemble
ADA Boosting	ADA	Ensemble - Boosting
Multi-Layer Perceptron	MLP	Neural Network
Decision Tree	DT	Decision
k-Nearest Neighbor	kNN	Nearest Neighbor
Logistic Regression	LR	Linear Model
Gradient Boosting	GB	Ensemble - Boosting
Hist Gradient Boosting	HGB	Ensemble - Boosting
Linear Discriminant Analysis	LDA	Discriminant Analysis
Complement Naive Bayesian	CNB	Probabilistic - Bayesian
Naive Bayesian	NB	Probabilistic - Bayesian

we evaluate the impact of dataset preprocessing techniques, such as class rebalancing and feature selection, both in isolation and in combination ($RQ_{3.B}$). Finally, we apply hyper-parameter tuning to the top five model configurations, identified on the basis of prediction accuracy, considering the algorithm, class rebalancing technique, and feature selection technique ($RQ_{3.C}$).

▷ 3A – *Model Training:* First, we train the machine learning models using their default parameters, as provided by the *scikit-learn* library,[6] without applying any dataset preprocessing. The dataset includes source code features for each method and coverage labels indicating whether a method is benchmarked. We evaluate the models using accuracy, precision, recall, F1 score, and balanced accuracy, as shown in Table 4 and discussed in Section 3.2.3. We employed 5-fold cross-validation to ensure a fair evaluation of our models. Specifically, the dataset is randomly split into five equal parts (folds), and in each of the five iterations, one fold is held out for testing while the remaining four are used for training. This process ensures that every data point is used exactly once for testing. Each experiment thus applies an 80-20 train-test split in a randomized manner, repeated five times with different partitions to mitigate sampling bias. Our study considers 11 machine learning models that are listed in Table 3. These models have been selected on the basis of their previous usage in software performance research area (Laaber et al. 2021). Below, we provide a description of each employed model:

Random Forest is an ensemble learning method that constructs multiple decision trees and combines their outputs for improved accuracy and stability. It reduces overfitting by averaging predictions across many trees, making it more robust than individual decision trees.

ADA Boosting (ADA Boost) is an ensemble method that combines multiple weak classifiers, usually decision stumps, to create a strong classifier. It assigns higher weights to misclassified instances, ensuring that subsequent models focus on difficult cases.

Multi-Layer Perceptron (MLP) is a type of artificial neural network with one or more hidden layers that use non-linear activation functions. It learns complex relationships between input and output through backpropagation and gradient descent.

[6] https://scikit-learn.org/

Decision Tree is a rule-based model that splits data into branches based on feature values, creating a tree-like structure for decision-making. While easy to interpret, it can overfit the data if not pruned.

k-Nearest Neighbor (k-NN) is a non-parametric algorithm that classifies data points based on the majority label of their k nearest neighbors in the feature space. It requires no training phase but can be computationally expensive for large datasets.

Logistic Regression is a statistical method used for binary classification that predicts the probability of an event occurring based on independent variables. It applies the logistic (sigmoid) function to map linear combinations of features into probability values.

Gradient Boosting is an ensemble technique that builds models sequentially, each correcting the errors of the previous one using gradient descent optimization.

Hist Gradient Boosting is an optimized version of Gradient Boosting that speeds up training by grouping feature values into discrete bins. This approach reduces memory usage and computation time while maintaining high accuracy.

Linear Discriminant Analysis (LDA) is a classification technique that projects high-dimensional data onto a lower-dimensional space while maximizing class separability. It assumes that each class follows a normal distribution and estimates parameters from training data. LDA is commonly used for dimensionality reduction and classification tasks.

Complement Naive Bayesian is a variation of Naive Bayes specifically designed for imbalanced text classification problems. It adjusts probability estimates to counteract class imbalances, making it more effective for handling rare categories.

Naive Bayesian is a probabilistic classification algorithm based on Bayes' theorem, assuming independence between features. It is computationally efficient and works well for high-dimensional data.

▷ (3B) – *Class Rebalancing:* Data obtained from the wrangling phase are used to assess the prediction capability of the considered models. Our dataset consists of 177,696 methods from 28 Java projects. Among these, 10,975 methods are benchmarked, while 166,721 methods are not. This represents a severe class imbalance, with benchmarked methods constituting only 6.58% of the total. To address this imbalance, we investigate two rebalancing techniques: oversampling and undersampling. For oversampling, we use the Synthetic Minority Over-sampling Technique (SMOTE) to increase the instances of the minority class (Chawla et al. 2002). SMOTE is a popular technique used during the training phase of machine learning models to handle imbalanced data. It generates new synthetic instances for the minority class by examining its nearest neighbors. For undersampling, we apply Random Undersampling (RUS) to reduce the instances of the majority class (Batista et al. 2004). RUS is a straightforward method that reduces the amount of data in the majority class to balance the dataset. This approach identifies the most common class and reduces its size to match the minority class one, thus resulting in a balanced dataset with fewer overall instances.

▷ (3B) – *Feature Selection:* This process includes removing co-linear and multi-co-linear features to avoid redundancy and improve model interpretability. We use Recursive Feature Elimination with Cross-Validation (RFECV) (Guyon et al. 2002) and Auto Spearman (Jiarpakdee et al. 2018) to identify the most relevant features for the prediction task. RFECV is a technique used during the feature selection phase of model training. It works by recursively considering progressively smaller sets of features. Starting with an ordered set of features ranked by importance, the less relevant features are pruned iteratively. This process continues until an optimal set of features is identified. By incorporating cross-validation, RFECV automatically finds the best set of features, thus ensuring that the selected features enhance model performance and generalization. Auto Spearman is another feature selection method that ranks features based on their Spearman correlation with the target variable. This approach helps in identifying features that have a strong relationship with the target, further refining the set of features used for prediction. We also investigate the contribution of the various features in predicting whether a method should be benchmarked or not. In particular, we used SHAP scores (SHapley Additive exPlanations), which are based on game theory to explain the output of machine learning models. The application of feature selection techniques resulted in the selection of 22 features using RFECV and 38 features using Autospearman, out of the 44 total source code features.

▷ (3B) – *Configuration Selection:* We consider various configurations by combining different algorithms and dataset preprocessing techniques. Each configuration is denoted as a tuple (a, cr, fs), where a represents one of the 11 algorithms in our study, cr indicates a class rebalancing method, and fs denotes a feature selection technique. Specifically, cr can be: no class rebalancing (original dataset), SMOTE, or RUS. Similarly, fs can be: no feature selection (original dataset), RFECV, or AutoSpearman. For each configuration, we train the model on the preprocessed training set and test it on the original (non-preprocessed) testing set. Our analysis involves training and testing 99 different configurations,[7] which we subsequently rank based on their F1 scores. We chose the F1 score because it balances the trade-off between false positives and false negatives, making it particularly useful in our context where both types of errors are significant.

▷ (3C) – *Hyperparameter Tuning:* We select the top-5 configurations as produced by the previous phase and apply hyperparameter tuning to them. Hyperparameter tuning involves adjusting the parameters of the machine learning models that are set before the learning process begins and are not learned from the data. We use RandomizedSearchCV[8] from the *scikit-learn* library to explore a range of hyperparameter values and find the best combination for the used models. In particular, RandomizedSearchCV helps identify the best set of hyperparameters that maximize the model performance. For tuning, we employ the F1 score as the objective metric. The results of the tuning process provided the final top-5 performing configurations for predicting whether a method is benchmarked.

▷ (3D) – *Transformer-Based Models:* To evaluate whether deep learning models pretrained on code can effectively predict performance-tested methods, we fine-tuned on our dataset two transformer-based encoders, i.e., CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2020). We adopted the defect detection setup introduced in CodeXGLUE (Lu et al. 2021), which defines a binary classification task for the source code. To

[7] The 99 configurations are derived from applying 11 models with 3 sampling methods and 3 feature selection techniques.

[8] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

prepare the inputs, we converted our raw method-level dataset into the required JSONL format, representing each method instance as a JSON object with a unique identifier as idx, including the raw method body with func, and specifying a binary label using target indicating whether the method was covered by performance tests. This resulted in three files: 'train.jsonl', 'val.jsonl', and 'test.jsonl', using an 80/10/10 split, respectively.

The fine-tuning process was performed as follows. Each Java method in our data set was tokenized as a sequence of subword tokens and prefixed with a special classification token [CLS]. The contextual embedding of this token was used as a holistic representation of the method. A fully connected layer followed by a sigmoid activation was added on top of the transformer output to perform binary classification, indicating whether the method was covered by a performance test.

Both models were trained using the same configuration: 5 epochs, 400 block sizes, 16 batch sizes, and a learning rate of 2×10^{-5}. We divided our data set into training sets (80%), validation sets (10%), and test sets (10%). These settings were chosen based on the configurations defined by the CodeXGLUE pipeline (Lu et al. 2021). CodeBERT operates solely on token sequences, while GraphCodeBERT extends this representation by incorporating data-flow edges to better capture structural relationships within the code.

Unlike traditional machine learning models evaluated in $RQ_{3.A-C}$, which relied on manually extracted features, these transformer-based models learned representations directly from raw source code. This approach enables the model to capture latent semantic and structural patterns without any hand-crafted feature design. Their training process and evaluation were fully integrated into our prediction pipeline, as illustrated in Fig. 1.

3.2 Employed Metrics

Our investigation utilized various metrics, each pertinent to a specific research question as shown in Table 4, and detailed below.

Table 4 Employed metrics

	Employed metrics	
RQ_1	*Coverage*	$C_T = \frac{\left\lvert\bigcup_{t\in T} M_t\right\rvert}{\lvert\mathcal{M}\rvert} \qquad \hat{C}_T = \frac{\left\lvert\bigcup_{t\in T} \hat{M}_t\right\rvert}{\lvert\mathcal{M}\rvert}$
	Overlap Ratio	$OR_T = \frac{\left\lvert\bigcup_{i,j\in T, i\neq j}(M_i \cap M_j)\right\rvert}{\left\lvert\bigcup_{k\in T} M_k\right\rvert}$
	Scope	$S_T = \frac{\sum_{t\in T}\lvert M_t\rvert}{\lvert T\rvert}$
RQ_2	*Total Execution Time*	$TET_T = \sum_{t\in T} e_t$
	Average Execution Time	$AET_T = \frac{\sum_{t\in T} e_t}{\lvert T\rvert}$
RQ_3	*Precision*	$\frac{TP}{TP+FP}$
	Recall	$\frac{TP}{TP+FN}$
	F1 Score	$\frac{2\times\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}$
	Balanced Accuracy	$\frac{\text{Recall of Class 1}+\text{Recall of Class 2}}{2}$

3.2.1 Coverage Metrics (RQ_1)

We rely on method-level coverage to assess the coverage of performance (or functional) testing suites. Specifically, we employ four metrics: *code coverage* C_T, *direct code coverage* $\hat{C}_T$, *overlap ratio* OR_T, and *scope* S_T.

- *Code coverage* (C_T) denotes the percentage of project methods covered–either directly or indirectly–by at least one test of the testing suite. Formally, a project method $m \in \mathcal{M}$ is considered as *covered* by a testing suite *T* if there exist at least one test $t \in T$ such that $m \in M_t$, where M_t denotes the set of methods invoked within the execution of *t*.
- *Direct code coverage* ($\hat{C}_T$) is defined as the percentage of project methods that are *directly covered* by at least one test of the testing suite. A project method *m* is considered as *directly covered* by a testing suite *T* if there exist at least one test $t \in T$ such that $m \in \hat{M}_t$, where $\hat{M}_t$ denotes the set of methods directly invoked by *t*.
- *Overlap ratio* (OR_T) measures the degree of redundancy within a testing suite *T*, by quantifying the extent of method coverage overlap across different tests. Table 4 provides a formal definition of this metric, where *i* and *j* denote two distinct tests that belong to *T*, M_i represents the set of methods covered by test i, and M_j represents the set of methods covered by test j. The numerator in OR_T denotes the number of methods covered in more than one test, while the denominator denotes the number of methods covered by the testing suite. OR_T values range from 0 to 1, where 0 indicates no overlap, i.e., each test cover distinct methods, and 1 indicates high test redundancy, i.e., all the methods are covered by more than one test.
- *Scope* (S_T) measures the average number of methods that an individual test covers within a testing suite. We use this metric because previous work has demonstrated that high coverage of tests (i.e., scope) tends to have a positive impact on the capabilities of uncovering performance issues (Ding et al. 2020).

3.2.2 Time Cost Metrics (RQ_2)

For each testing suite T, we consider two time cost metrics: *total execution time* (TET_T) and *average execution time* (AET_T). The *total execution time* (TET_T) is the cumulative sum of the execution times for all tests in the suite, i.e., the time required to run the entire testing suite. The *average execution time* (AET_T) is calculated as the mean execution time of the individual tests within the suite.

3.2.3 Model Performance Metrics (RQ_3)

To evaluate the prediction accuracy of our machine learning models, we employ well-established metrics for binary classification: *precision*, *recall*, *F1 score*, and *balanced accuracy*. We specifically choose *balanced accuracy* over traditional *accuracy* because the latter can be misleading in the presence of class imbalance (Brodersen et al. 2010).

4 Results and Discussion

In this section, we present and discuss the results of our analysis. As illustrated in Table 5, we analyzed 2,190 JMH benchmarks from 28 software projects, and 256,233 JUnit tests from 19 software projects. For RQ1 and RQ2, we analyze performance tests on all 28 projects (both for coverage and time cost). However, 9 Projects were excluded from the comparison of performance and functional testing suites due to technical issues encountered during the JUnit data collection (see Section 3.1 for details).

4.1 RQ_1: *To What Extent Do Performance Tests Cover the Source Code of Software Systems?*

To answer RQ_1, we analyze the coverage of JMH benchmarks within our dataset of 28 software projects. We centre our analysis around two key metrics: (i) *Benchmark Coverage*

Table 5 Java systems overview

Project	GitHub Stars	Methods (Total)	Benchmarks (Total)	Unit Tests (Total)	Domain
arrow	12600	3789	34	869	Analytics Tools
byte-buddy	5800	5046	39	6357	Code Generation
cantaloupe	259	3475	103	3063	Computer Graphics
client_java	2100	386	33	217	JVM Tools
commons-bcel	223	3132	3	137	JVM Tools
crate	3800	24155	39	–	Database Systems
eclipse-collections	2300	15219	515	–	Programming Utility
fastjson	25500	17524	4	4979	Parsing Library
feign	9100	979	8	913	Web Development
HdrHistogram	2100	780	12	147	Analytics Tools
imglib2	278	3432	25	635	Computer Graphics
iri	1200	1554	3	398	Data Structures
jdbi	1800	2379	76	1428	Database Systems
jetty.project	3700	18060	48	–	Web Development
jgrapht	2400	3782	51	2416	Programming Utility
jooby	1600	3331	3	485	Web Development
kafka	26000	16157	27	–	Data Streaming
logbook	1600	811	20	564	Web Development
netty	31900	16615	221	–	Network Applications
objenesis	568	207	13	45	Programming Utility
panda	247	1479	4	61	Programming Utility
protostuff	2000	3312	16	–	Programming Utility
r2dbc-h2	191	291	8	259	Database Systems
rdf4j	331	14041	14	–	Database Systems
RxJava	47300	8282	217	–	Programming Utility
SquidLib	439	5663	236	73	Computer Graphics
tinkerpop	1800	7753	57	–	Database Systems
vert.x	13800	5685	41	4095	JVM Tools

(C_T), which measures the extent of method coverage by JMH benchmarks, and it includes *direct coverage* ($\hat{C}_T$); (ii) *Overlap Ratio* (OR_T), which assesses the degree of redundancy in method coverage across different benchmarks of the same project. Besides this, we analyze the coverage of JUnit tests and subsequently compare it with the coverage of JMH benchmarks within a subset of 19 software projects, as mentioned above.

Benchmark Coverage Our analysis revealed that the coverage by JMH benchmarks in software projects is relatively low compared to the total number of methods: C_T averaged 8.8% with 2.1% of methods directly covered ($\hat{C}_T$) across all 28 projects.

Figure 3 shows the distribution of benchmark coverage (C_T) across the examined projects. The y-axis represents the percentage of methods benchmarked in each project, while the x-axis enumerates the projects. The project with the highest coverage is panda, where 48.82% of methods are covered. On the other hand, the jooby project has the lowest coverage (0.27%). The analysis indicates diverse levels of coverage among the systems with a standard deviation of 9.2%. By excluding the panda project, considered as an outlier, from our analysis, the benchmark coverage's standard deviation was 4.9%. This variation can be attributed to differences in the size of the projects (in terms of their number of methods, benchmarks and unit tests), to the particular development and testing practices adopted, and to the specific performance requirements and constraints. It is evident from the figure that panda is an outlier. Perhaps, in panda, we can associate the high coverage of benchmarks with the nature of the project, which is a lightweight and extensible programming language

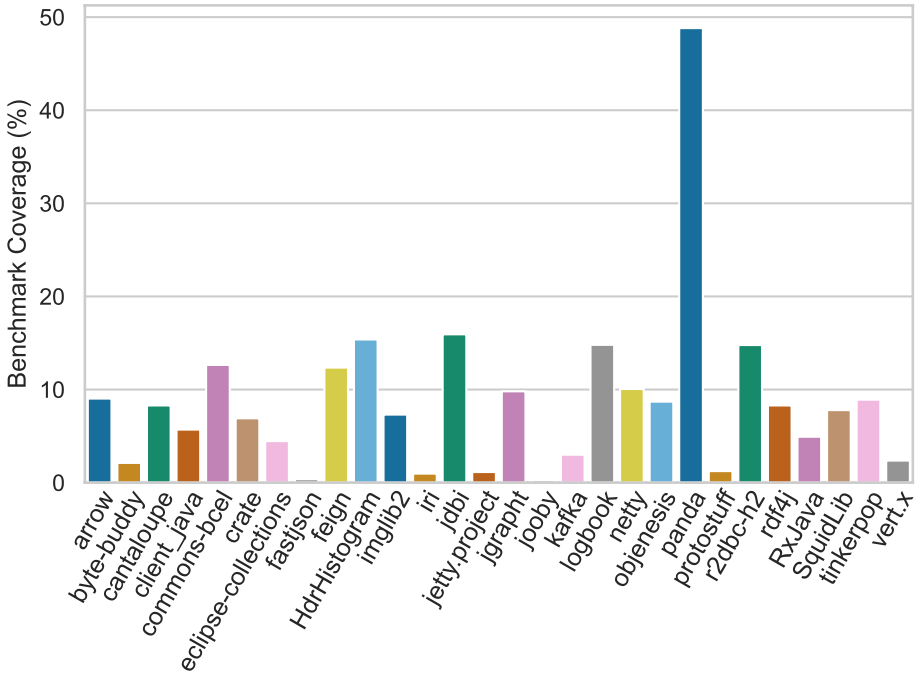

Fig. 3 Coverage (C_T) of benchmarks across projects

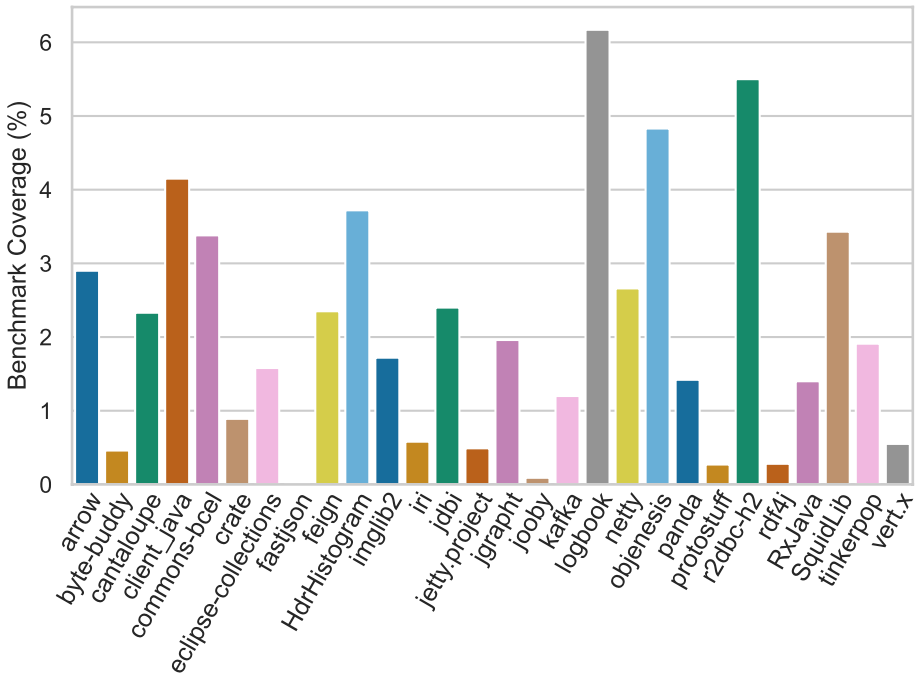

Fig. 4 Direct coverage ($\hat{C_T}$) of benchmarks across projects

toolkit. Due to required efficiency and scalability, developers may focus extensively on microbenchmarking.

A detailed look at *direct benchmark coverage*, depicted in Fig. 4, reveals a similar trend of varied coverage. As a first consideration, the average direct benchmark coverage across many projects is around 2%, substantially lower than the overall benchmark coverage. The figure also shows a sensibly lower variation in the direct coverage compared to the overall coverage. The standard deviation for direct benchmark coverage is 1.7%, which also indicates a lower spread than the one in overall benchmark coverage. The observed difference between direct and indirect coverage may be explained by developers' tendencies to target high-level methods during performance testing. Such high-level methods often result in a large number of indirect invocations, which might broaden the overall benchmark coverage.

Overlap Ratio Figure 5 illustrates the degree of overlap in method coverage, which represents the redundancy in method coverage across different benchmarks in the project.

The overlap ratio in the considered projects ranges from a minimum of 0.09 in commons-bcel to a maximum of 1 in byte-buddy. In general, we can observe relatively high overlaps, with an average of 74% methods covered by more than one benchmark. While this average suggests a high degree of redundancy within performance testing suites, this may also reflect the attempts of developers to test a method under various workloads (Samoaa and Leitner 2021). Nonetheless, our analysis highlights room for improving the efficiency of performance testing by reducing unnecessary overlap.

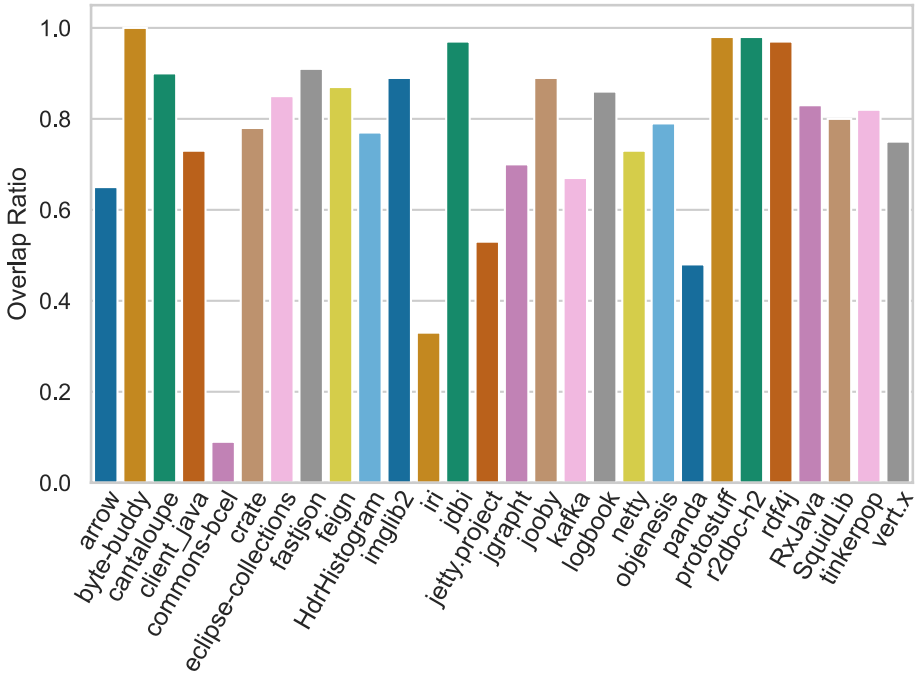

Fig. 5 Overlap Ratio (OR_T) of benchmarks

Comparison of JMH Benchmarks and JUnit Tests Coverage Figure 6 illustrates a comparison between the coverage achieved by performance testing and the functional testing one. Clearly, the percentage of methods covered by JUnit tests tends to be much higher than the one of JMH benchmarks, thus indicating a broader coverage for functional testing in the projects under study. On average, the coverage achieved by a performance testing suite is a quarter of that achieved by a functional testing suite (10.4% *versus* 41.3%). The JUnit tests coverage is significantly higher in many projects (like arrow, cantaloupe, fastjson, iri, jooby, jgrapht, and vert.x) as compared to the JMH benchmarks coverage. However, it is also interesting to note an exception, i.e., the SquidLib project, where not only the JUnit Test coverage is relatively low, but also the JMH benchmark coverage exceeds the JUnit tests one. Upon closer examination of the project code, we attributed this anomaly to the performance-driven focus of the project. Specifically, SquidLib is a Java library crafted to serve as a comprehensive toolkit for the development of various gaming applications. This orientation likely leads to a higher priority being placed on performance testing over unit testing, thereby resulting in a more extensive benchmark coverage.

Similarly, Fig. 7 provides a comparison of direct coverage between JMH benchmarks and JUnit tests across the same set of projects. The trend is similar to the previous one. Few exceptions appear also here, such as (again) SquidLib, where direct benchmark coverage exceeds the direct coverage by unit tests. We also note substantial differences across projects in terms of the direct coverage of methods by both JMH benchmarks and JUnit tests. For instance, in the r2dbc-h2 project, the direct coverage of JUnit tests reaches 30%, whereas

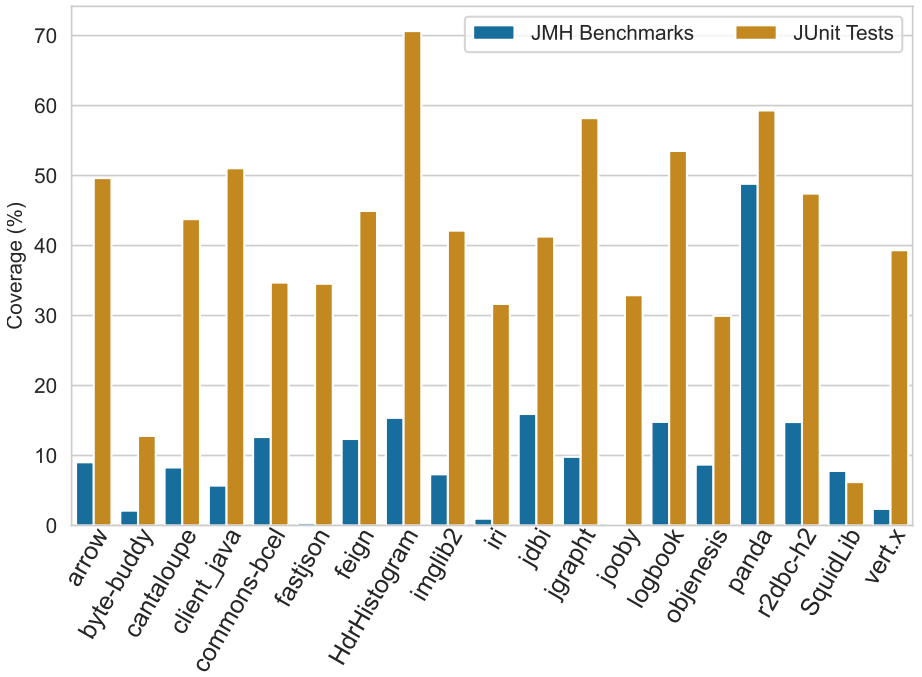

Fig. 6 Comparison of coverage (C_T) of JMH Benchmarks and JUnit Tests across projects

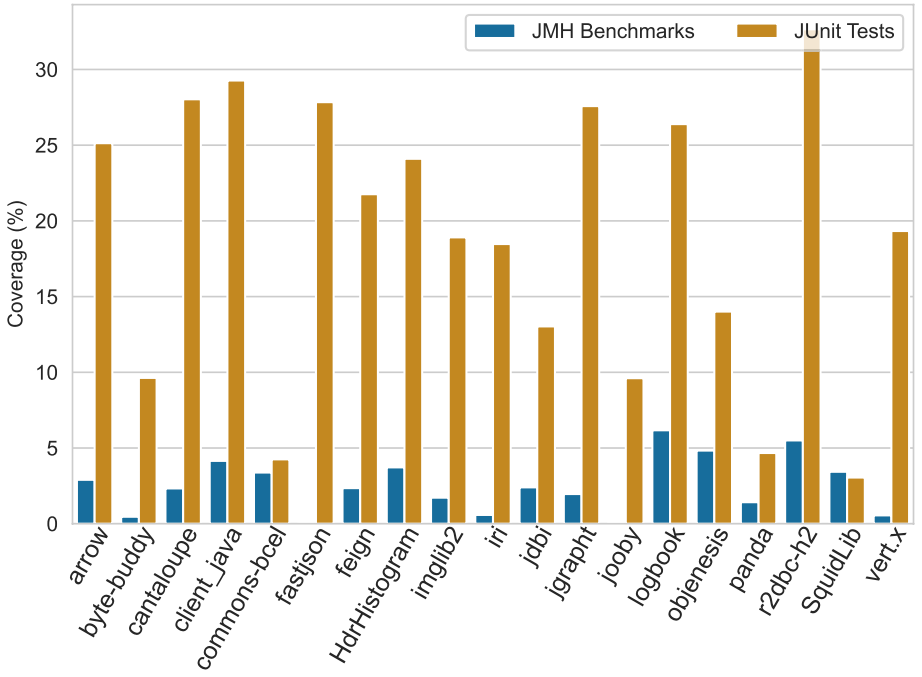

Fig. 7 Comparison of direct coverage ($\hat{C_T}$) of JMH Benchmarks and JUnit Tests across projects

the one of JMH benchmarks is nearly 5%. This observation emphasizes the idea that, while functional tests point to broad coverage to ensure overall correctness, performance benchmarks often target specific, performance-sensitive parts of the code.

Overlap Ratio Comparison Our analysis reveals that JMH benchmarks exhibit a more significant overlap in coverage compared to JUnit tests one. In particular, we found that JMH suites show higher overlap in coverage compared to that of the JUnit suites in 68% of the projects. As shown in Fig. 8, byte-buddy, r2dbc-h2, and jdbi exhibit near-complete overlap for JMH benchmarks, indicating extensive redundancy in performance testing. On the other hand, commons-bcel stands out with a much lower overlap ratio for JMH benchmarks (0.09), yet a higher overlap for JUnit tests (0.66), indicating a different testing approach. Meanwhile, fastjson has a high overlap in JMH benchmarks (0.91), but a much lower ratio in JUnit tests (0.15). These variations highlight the different testing priorities and strategies between performance and functional correctness across projects.

Scope Comparison of JMH Benchmarks and JUnit Tests Figure 9 compares the scope (S_T) of JMH benchmarks and the JUnit tests one. It is interesting to observe that, while the coverage of performance testing suites is generally lower than the functional testing suites one, the average coverage (i.e., scope) of individual benchmarks is notably larger. Specifically, JMH benchmarks show, on average, approximately three times larger scope than their JUnit counterparts. This significant difference in the scope is more evident in projects like panda and commons-bcel, likely due to the project-specific nature of JMH benchmarks, which

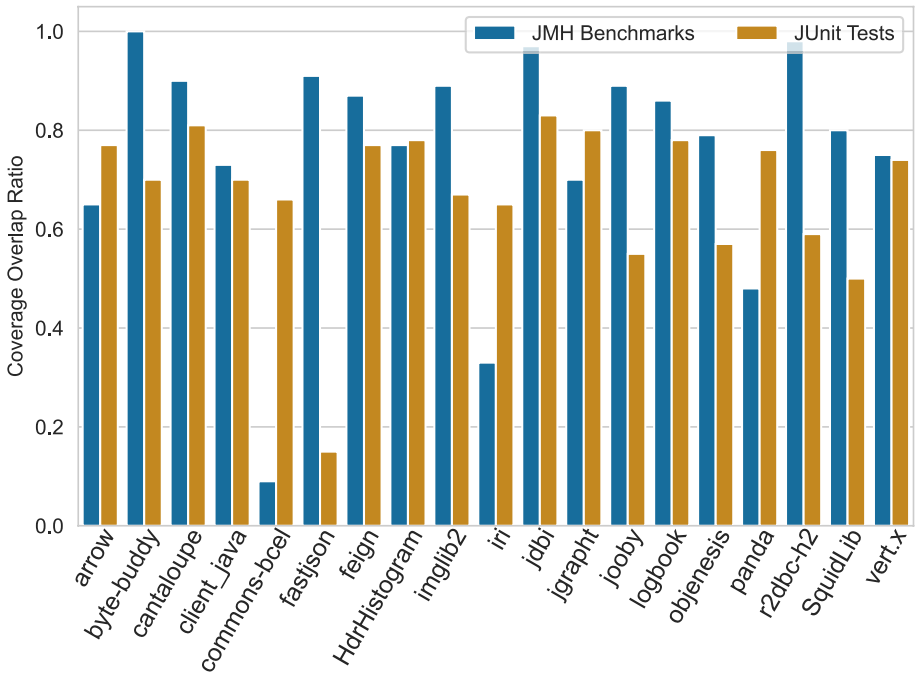

Fig. 8 Comparison of Overlap Ratio coverage ($O\hat{R}_T$) of JMH Benchmarks and JUnit Tests across projects

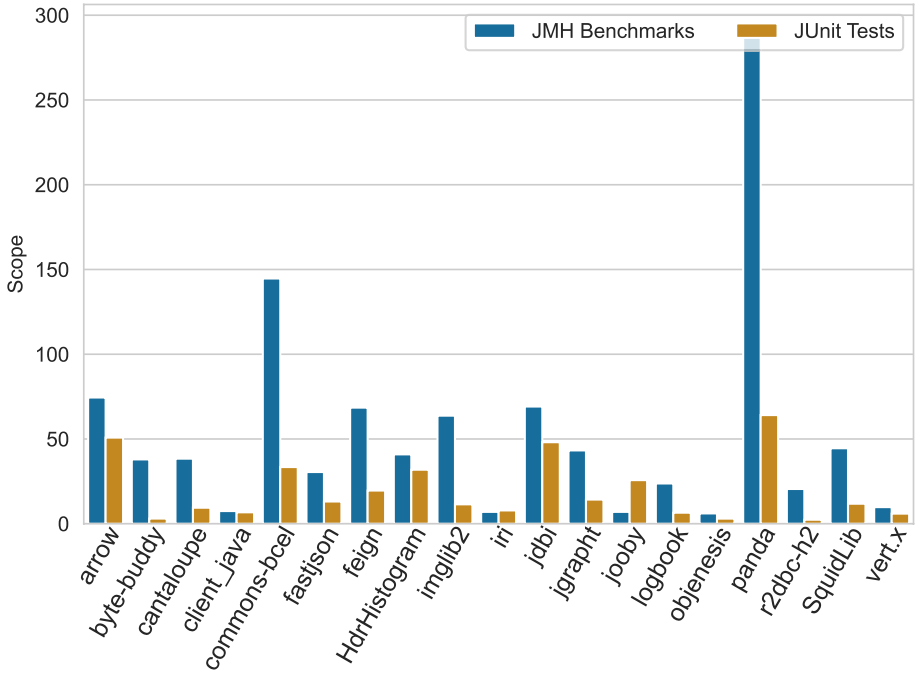

Fig. 9 Scope (S_T) of JMH Benchmarks vs. JUnit Tests

emphasize testing performance under various configurations and workloads. These results, once again, suggest that JMH benchmarks tend to target high-level methods, which leads to a larger number of method invocations, whereas JUnit tests tend to target lower-level methods. This finding is in line with our previous observation about the difference between direct and indirect coverage.

> **Answer to RQ$_1$:** The analyzed projects show that performance testing suites achieve a limited code coverage of 8.8% on average, which is a quarter of the code coverage achieved by functional tests. Additionally, in 68% of the projects, performance testing suites exhibit higher overlap than functional testing suites, thus indicating larger redundancy in the tested methods.

4.2 RQ$_2$: *What is the Time Cost of Performance Tests?*

In this subsection, we present our findings on the execution time of JMH benchmarks and compare them to the functional testing ones.

Total Execution Time Figure 10 displays the execution time of performance testing suites. On the y-axis, we report the total execution time (TET_T) in hours, using a logarithmic scale. The black dashed line depicts the overall average time, which is about 29.3 hours. The TET_T distribution varies significantly from one project to another, with some testing

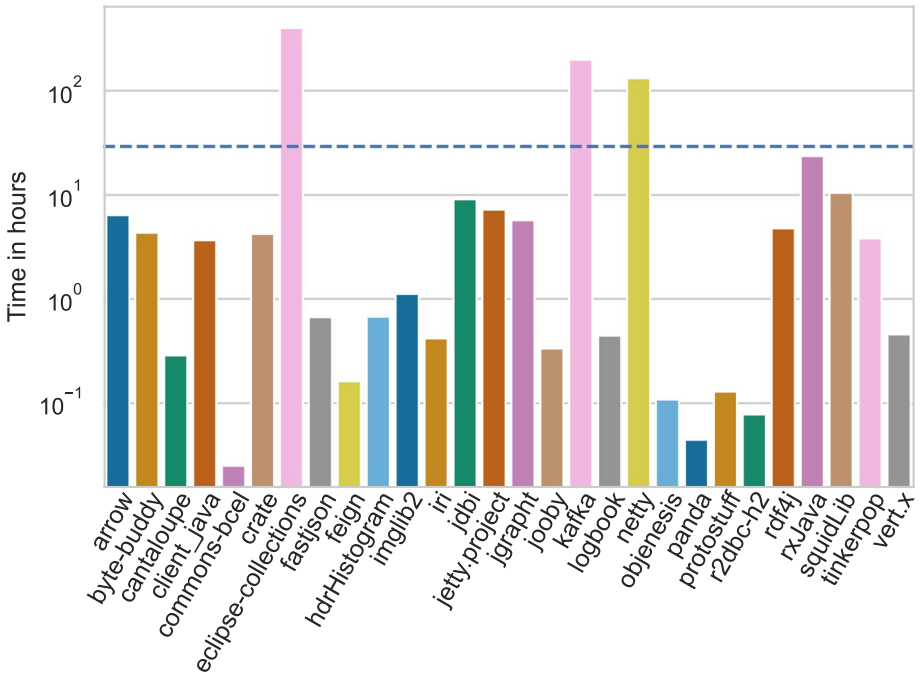

Fig. 10 JMH total execution time (TET_T)

suites completing in just a few minutes, while others requiring over 100 hours of execution. The less time-consuming suite is the commons-bcel one, which required only 90 seconds to run its benchmarks. Along with commons-bcel, only other two projects kept the TET_T under 6 minutes (i.e., 10^{-1} hours), namely panda and r2dbc-h2. About the most time-consuming performance testing suites, few ones exceeded 10 hours. Interestingly, two projects (i.e., squidLib and rxJava) exceeded 10 hours but did not overcome the threshold of 100 hours, which was required by other three projects. In particular, eclipse-collection required 401 hours, thus representing the most time-intensive project for performance testing.

Average Execution Time In Fig. 11, the bar chart depicts the average execution time of benchmarks (AET_T) for each performance testing suite. The dashed line shows the average AET_T across projects, which is about 23 minutes. A close examination of this chart confirms that the diversity across projects observed for TET_T also holds for AET_T. In particular, commons-bcel, which was the least time consuming one in terms of TET_T, resulted in a relatively low AET_T of 30 seconds, and similarly panda and r2dbc-h2 have maintained a low average execution time. Contrariwise, eclipse-collections (515 benchmarks and an AET_T of 2,805 seconds) and netty (221 benchmarks and an AET_T of 2,149 seconds) are very time-consuming projects, as equipped with a high amount of benchmarks.

The most time-consuming performance testing suite is the one of kafka, with an AET_T of about 7 hours per benchmark, followed by eclipse-collections. Interestingly, the perfor-

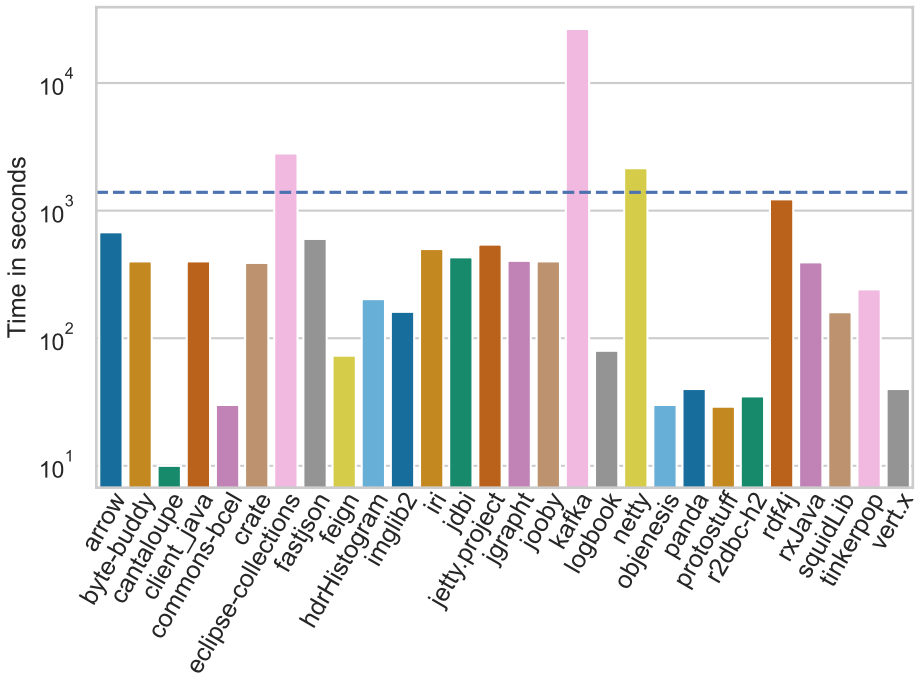

Fig. 11 JMH average execution time (AET_T)

mance testing suite of kafka consists of a limited number of benchmarks (i.e., 27), each of which is then notably time-consuming. We investigated the JMH reports to understand the reasons behind these high AET_T values, and we discovered that the likely reason is the large number of repetitions of kafka benchmarks under different input values.

This allows the benchmark to measure performance across a variety of configurations. Each configuration results in an additional execution of the benchmark, which naturally increases the total execution time (Samoaa and Leitner 2021).

Total Execution Time Comparison In Fig. 12, we compare TET_T of JMH and JUnit testing suites. As expected, the analysis revealed that, in general, performance testing is significantly more time-consuming than functional testing. For instance, by examining SquidLib, the most time-consuming project for performance testing, we observe that TET_T for the JMH suite is exponentially higher than the JUnit suite one (37,735 seconds *versus* 18.5 seconds). Interestingly, the least time-consuming project in terms of functional testing is vert.x that requires 821 seconds, whereas TET_T for its JMH suite is 1,640 seconds, i.e., more than twice the JUnit test suite execution. We can notice comparable results if we analyze less time-consuming projects. For instance, commons-bcel requires 90 seconds for executing the whole JMH suite and reports a TET_T of 15.27 seconds for the JUnit suite. Another interesting case is objenesis, which requires half a second for executing the JUnit testing suite, and 390 seconds for executing the JMH suite. Performance testing suites have, on average, an execution time cost 62 times higher than functional testing suites one.

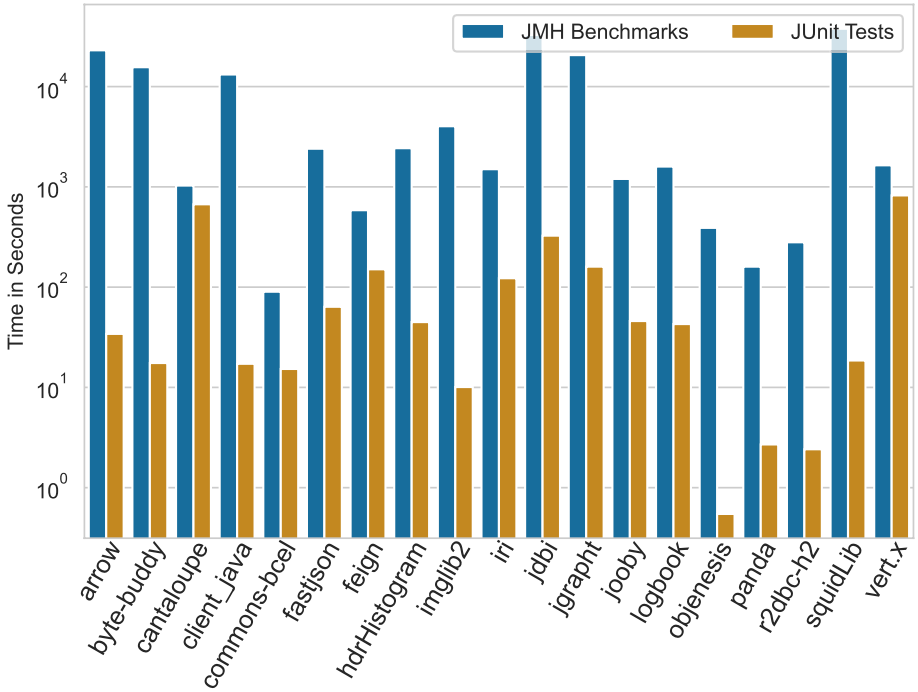

Fig. 12 Total execution time (TET_T) comparison

Average Execution Time Comparison There is a significant difference in test-level execution time between JMH and JUnit tests across all projects. We found that JMH benchmarks exhibit an average AET_T of approximately 245.9 seconds. On the other side, the average AET_T of JUnit tests does not exceed 0.098 seconds. On average, an individual JMH benchmark demands about 2,507 times the execution time required by a JUnit test. This substantial difference can primarily be attributed to the inherent characteristics of performance testing, which require repeated executions to achieve reliable results (Maricq et al. 2018; Mytkowicz et al. 2009). In fact, these repeated executions are typically used to address two main factors: (i) the inherent variability and fluctuations in performance measurements (Maricq et al. 2018; Georges et al. 2007; Kalibera and Jones 2013), and (ii) the need to achieve a steady state performance condition that closely resembles real world production environments (Traini 2022; Barrett et al. 2017). As highlighted in previous studies, there is often a trade-off between the quality of performance testing results and the time cost of their execution (Kalibera and Jones 2013; Laaber et al. 2020; Traini et al. 2024). Naturally, this issue does not apply to functional tests, which typically require only a single execution to verify the correctness of program behavior.

We report the complete results related to AET_T of JUnit tests in our replication package (Imran et al. 2024).

Answer to RQ$_2$: In the analyzed projects, performance testing suites have, on average, a time cost 62 times higher than that of functional testing suites. Specifically, the average execution time for a performance testing suite is 29.3 hours, while individual performance tests last 23 minutes on average. These results highlight the significant execution time cost of performance testing, particularly when compared to the much shorter execution times of functional testing.

4.3 RQ$_3$: *Are There any Specific Characteristics in the Source Code that can Guide Performance Testing?*

To address RQ$_3$, we examine the relationship between source code features and the likelihood of being covered by performance tests. Our goal is to search for peculiarities in the methods selected for performance testing. To achieve this, we investigate the possibility of employing machine learning models to predict whether a method should be performance-tested based on its source code features.

4.3.1 RQ$_{3.A}$: *Can we Predict Whether a Method Should be Performance Tested using Machine Learning Models with Default Settings?*

We first analyze the prediction accuracy of 11 classification algorithms (discussed in Section 3.1.3) using their default settings.

Figure 13 presents the results for the employed machine learning models in terms of various performance metrics, including precision, recall, F1 score, and balanced accuracy, as detailed in Table 4. Each subplot corresponds to a different performance metric used to evaluate prediction performance. The results indicate that it is challenging to predict with high accuracy which methods should be benchmarked based on source code features. The F1 score, which is particularly meaningful in this context, shows that the best models barely reached 0.4%, with none of the tested models exceeding 0.5%. Furthermore, only the Decision Tree, k-Nearest Neighbors, and Random Forest models achieved reasonable performance levels. In contrast, the Naïve Bayesian (NB) model performed the worst, remaining below 0.2%.

As shown in Fig. 13, the Naïve Bayesian model demonstrates a high recall but very low precision. This indicates a higher rate of false positives, meaning that while the NB model is effective at identifying methods that should be benchmarked (high recall), it also incorrectly flags many unsuitable methods as suitable (low precision). This leads to a higher rate of false positives, thus making the model less reliable when it is crucial to ensure that most of the predicted positives are true positives.

On the other hand, the Gradient Boosting (GB) and Histogram Gradient Boosting (HGB) models perform apparently well in terms of precision. This score is due to the small amount

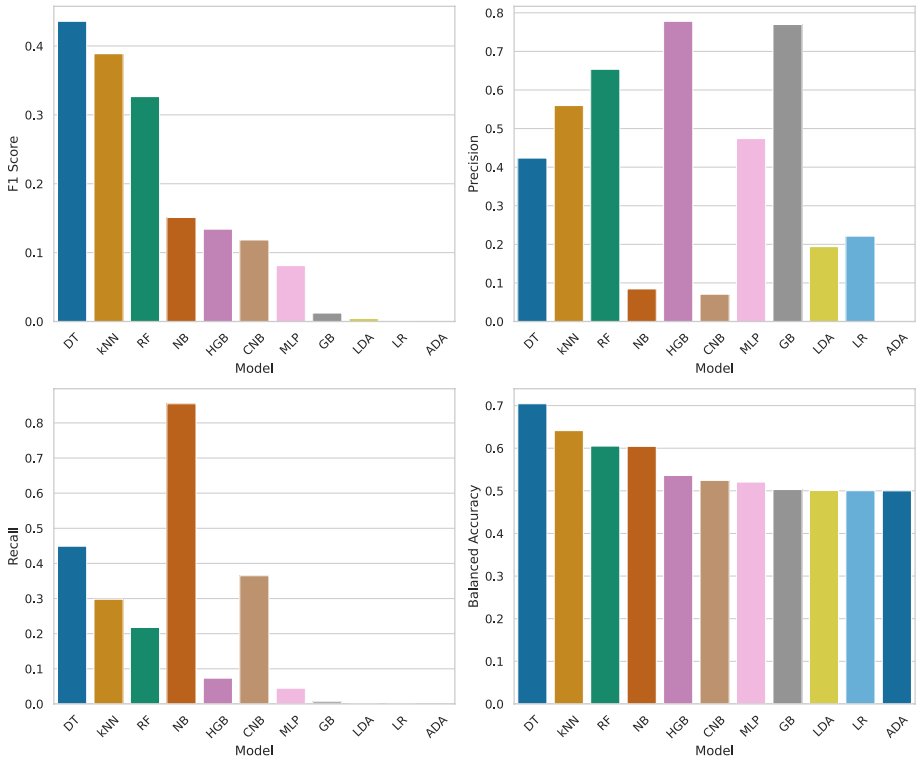

Fig. 13 Predictive performance of static source code features

of true and false positive predictions. However, if we consider their recall, then we notice values close to 0, thus confirming that most of the predictions were true negative. This fact is proved also by the F1 score values, close and lower than 0.1%.

F1 score balances precision and recall, thus providing a comprehensive metric to evaluate models. Consequently, the Decision Tree (DT), k-Nearest Neighbors (kNN), and Random Forest (RF) models, with their higher F1 scores, appear the most effective ones for predicting whether a method should be benchmarked based on static features of the source code. However, it is important to note that the absolute F1 scores of these models are quite low, thus implying significant room for improvement in their prediction capabilities.

Since we are dealing with a relevantly unbalanced dataset, as reported by the coverage analysis, we have also employed the Balanced Accuracy metric. The results of this latter metric show that, in all the examined cases, the models can correctly identify the true negatives (i.e., methods that are correctly classified as not benchmarkable). This result is due to the evident unbalanced situation. In other words, the models easily identify the non-benchmarkable methods because, during the training phase, most of the methods were not covered.

4.3.2 $RQ_{3.B}$: *How does the prediction accuracy change when employing feature selection and class-rebalancing techniques?*

To answer $RQ_{3.B}$, we investigate whether and to what extent pre-processing steps impact the model prediction performance. To do this, we apply a combination of two commonly

used pre-processing steps: *(1)* class-rebalancing using the Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al. 2002) and Random Under-Sampling (RUS) (Batista et al. 2004), and *(2)* removal of co-linear and multi-co-linear features using AutoSpearman (Jiarpakdee et al. 2018) and Recursive Feature Elimination with Cross-Validation (RFECV) (Guyon et al. 2002).

Figure 14 illustrates the prediction performance metrics for all studied algorithms across various combinations of these pre-processing techniques. We assess nine distinct configurations: *(1)* a baseline without feature selection and class rebalancing, *(2)* RFECV without rebalancing, *(3)* AutoSpearman without rebalancing, *(4)* no feature selection with SMOTE class rebalancing, *(5)* no feature selection with RUS, *(6)* RFECV combined with SMOTE, *(7)* AutoSpearman combined with SMOTE, *(8)* RFECV combined with RUS, and *(9)* AutoSpearman combined with RUS. This comprehensive approach results in 99 distinct configurations (11 models $\times$ 9 pre-processing combinations), thus allowing us to thoroughly evaluate the impact of various pre-processing techniques on model performance.

Each subplot in Fig. 14 represents the F1 scores of a specific machine learning model across different combinations of feature selection and sampling techniques. Significant variations in model performance across different pre-processing configurations can indeed be observed. First of all, the effect of removing co-linear and multi-co-linear features using

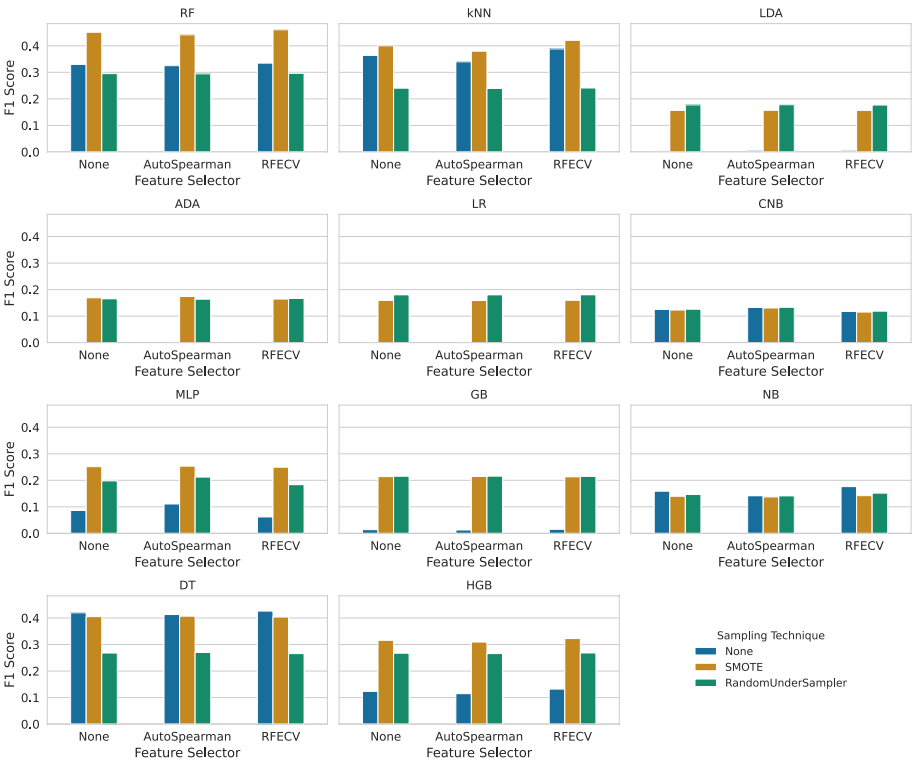

Fig. 14 F1 scores for all models with sampling methods and feature selection techniques

RFECV and AutoSpearman is a notable variation across models. For example, Random Forest (RF) always shows a slight improvement in F1 score when RFECV is applied (from 0.326 to 0.335 without sampling). It has to be noticed that, when combined with SMOTE, RF's performance increases to reach an F1 score of approximately 0.46. This indicates that RF benefits from both feature selection and class rebalancing in general (specifically RFECV and SMOTE).

The subplot for k-Nearest Neighbors (kNN) in Fig. 14 shows that the baseline F1 score without any pre-processing is better than the one obtained by using AutoSpearman for feature removal. In addition, the usage of RFECV does not provide any significant improvement. The combination of RFECV and SMOTE results in an F1 score of 0.420, which, while higher than the baseline, does not provide any relevant improvement, as the the F1 Score using only SMOTE is still 0.420.

For Decision Trees (DT), the highest baseline F1 score is 0.435. The application of RFECV alone slightly reduces the score to 0.426, while AutoSpearman lowers it further to 0.413. Combining RFECV with SMOTE performs relatively better, but the baseline is still more effective. However, for DT, SMOTE also occurs to be more effective than RUS with all the feature pre-processing techniques.

In general, models such as Logistic Regression (LR), Gradient Boosting (GB), and Linear Discriminant Analysis (LDA) show improvements with pre-processing. The baseline F1 score for LR is nearly zero, and the application of RUS alone provides the highest F1 score. This shows the effect of class rebalancing on the performance improvement of this model.

For Naive Bayes (NB), the application of RFECV alone provides the highest F1 score of 0.187, whereas Complement Naive Bayes (CNB) exhibits relatively stable performance across different pre-processing configurations. CNB shows minimal variation in performance, with F1 scores remaining around 0.113 to 0.118.

To gain insights into the influence of individual features, we computed SHAP values. The results are presented in Fig. 15. We employed the Decision Tree model for this analysis, as it showed the highest predictive performance under default settings (i.e., without sampling, hyperparameter tuning, or feature elimination). The SHAP analysis reveals that most features have a relatively low impact on the model's predictions regarding whether a method should be benchmarked. Interestingly, the features eliminated through RFECV largely coincide with those having the lowest SHAP values, further validating the effectiveness of our selection process.

Therefore, in answering $RQ_{3.B}$, our analysis shows that the impact of removing co-linear and multi-co-linear features, as well as applying class-rebalancing techniques, significantly varies across different machine learning models. Feature selection methods like RFECV and AutoSpearman exhibit model-specific effects, with some algorithms benefiting more than others. Class-rebalancing techniques, particularly SMOTE, generally show improvement in model performance for algorithms like Random Forest and k-Nearest Neighbors. However, Decision Trees perform better with the baseline configuration, while models like Logistic Regression and Naive Bayes show only incremental improvements. Thus, the combination of feature selection and class-rebalancing does not always yield additive benefits.

It is important to note that, despite these pre-processing steps, the overall prediction performance based on the static source code features remains relatively low across all

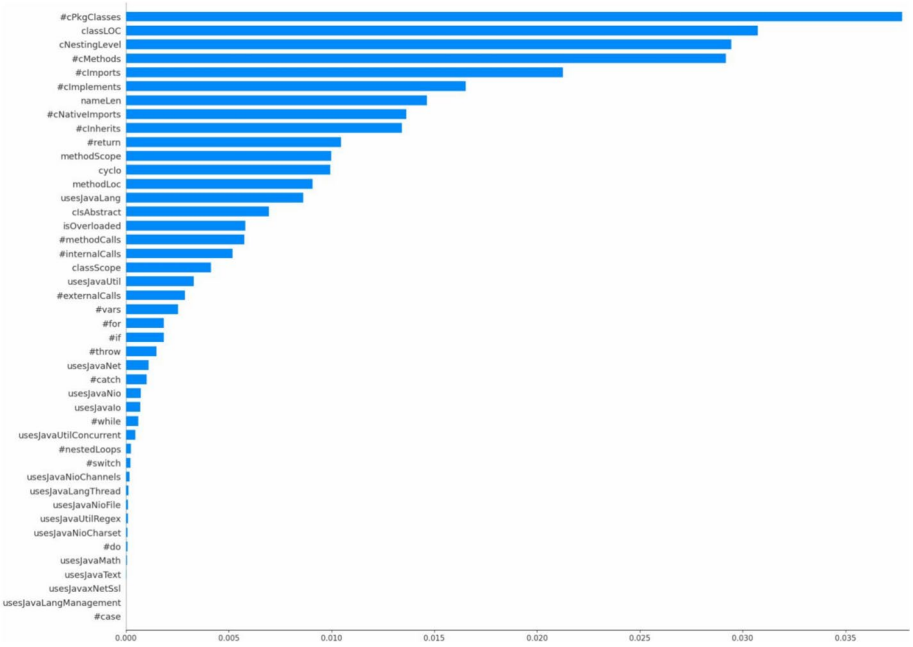

Fig. 15 SHAP values calculated using the decision tree model

models and configurations. The highest F1 score achieved is only 0.467, which indicates that there is still significant room for improvement in predicting the likelihood of a method being tested for performance. This suggests that while pre-processing steps can enhance model performance to some extent, they are not sufficient to overcome the challenges in this prediction task.

4.3.3 $RQ_{3.C}$: *How Does the Prediction Accuracy Change After Tuning the Hyperparameters of Machine Learning Models?*

Based on findings from $RQ_{3.A}$ and $RQ_{3.B}$, in this subsection, we answer $RQ_{3.C}$ by selecting the top 5 configurations with the highest F1 scores from the previous analysis (see Table 6). We then apply hyperparameter tuning to the respective models in these top 5 configurations to evaluate the effect on prediction performance (see Table 7).

By comparing Tables 6 and 7, it is evident that hyperparameter tuning resulted in moderate improvements in model performance. The best F1 score increased slightly from 0.46 to

Table 6 Top 5 configurations based on F1 score without hyperparameter tuning

Model	Sampling Technique	Feature Selector	Precision	Recall	Balanced Accuracy	F1 Score
RF	SMOTE	RFECV	0.56	0.40	0.69	0.46
RF	SMOTE	None	0.55	0.40	0.69	0.46
DT	None	None	0.42	0.45	0.70	0.44
DT	SMOTE	None	0.42	0.44	0.70	0.43
kNN	SMOTE	RFECV	0.30	0.72	0.80	0.42

Table 7 Top 5 configurations based on F1 score with hyperparameter tuning

Model	Sampling Technique	Feature Selector	Precision	Recall	Balanced Accuracy	F1 Score
RF	SMOTE	RFECV	0.56	0.40	0.69	0.47
RF	SMOTE	None	0.56	0.40	0.67	0.47
kNN	SMOTE	RFECV	0.38	0.57	0.75	0.46
DT	None	None	0.44	0.45	0.70	0.44
DT	SMOTE	None	0.42	0.44	0.70	0.43

0.47 for the configuration of Random Forest with SMOTE and RFECV. Also, Random Forest maintained its position as the best-performing model both before and after hyperparameter tuning, which shows the relatively robust configuration for predicting the likelihood of the method to be covered by performance testing. Interestingly, while the top two configurations remain consistent (RF with SMOTE), we observe changes in the subsequent rankings. The k-Nearest Neighbors model with SMOTE and RFECV shows some improvement, with its F1 score increasing from 0.42 to 0.46 after tuning. Similarly, hyperparameter tuning affected precision and recall differently across models. For instance, the baseline configuration of Decision Tree model (no sampling and no feature selection) had an improvement in precision (from 0.42 to 0.44) with a stable trend on recall. Most models showed a slight increase in Balanced Accuracy after hyperparameter tuning with the exception of kNN, where it decreased from 0.80 to 0.75.

Therefore, the application of hyperparameter tuning has not considerably improved the overall prediction performance, with the highest F1 score reaching only 0.48. We observe that while hyperparameter optimization improves the performance of some models with certain configurations compared to others, it still lacks promising results for this specific task of identifying methods that should be benchmarked based on static code features. This also highlights that predicting which methods should be benchmarked based on static code features remains a challenging task.

4.3.4 $RQ_{3.D}$: *Can Transformer-based Models, such as CodeBERT and GraphCodeBERT, Improve the Prediction Accuracy Compared to Traditional Machine Learning Models?*

To further investigate the prediction of methods that should be tested for performance, we fine-tuned two transformer-based models, CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2020), using the CodeXGLUE defect detection pipeline (Lu et al. 2021). The goal was to evaluate whether deep learning models pre-trained on source code can outperform traditional machine learning models trained on static features.

Table 8. reports the prediction performance of CodeBERT and GraphCodeBERT. Both models achieved high accuracy (above 94%), CodeBERT achieving an F1 score of 0.23 and GraphCodeBERT achieving 0.34. The improvement with GraphCodeBERT was primarily

Table 8. Prediction performance of transformer-based models in identifying performance-tested methods

Model	Accuracy	Precision	Recall	F1-score
CodeBERT	0.94	**0.78**	0.14	0.23
GraphCodeBERT	**0.95**	0.73	**0.22**	**0.34**

The values highlighted in bold represent the highest numbers in their respective columns

due to a more balanced trade-off between precision and recall, which suggests that incorporating structural contexts, such as data flow graphs, can enhance generalization when learning from raw source code.

Although these models did not outperform the best traditional ML configuration (Random Forest with SMOTE and RFECV, F1 = 0.47), they offer an effective alternative by eliminating the need to engineer manual features and learn directly from the source code. In particular, GraphCodeBERT's superior F1 score compared to CodeBERT highlights the value of incorporating structural code information, such as data flow edges, into the representation. These findings reinforce the growing relevance of learning-based representations for code, in line with recent neural architectures like HGNN4Perf (Fu et al. 2024), which use hypergraph structures to capture complex performance-related dependencies. Altogether, our results suggest that while transformer-based models alone may not yet outperform well-tuned feature-based classifiers, they hold promise as a foundation for future methods that integrate richer program semantics.

Answer to RQ_3: In the analyzed projects, machine learning models exhibit limited accuracy in predicting whether a method is subject to performance testing, by achieving a top F1 score of 0.44 with default settings. Even after applying well-knwon practices, such as feature selection, class rebalancing, and hyper-parameter tuning, the best model only achieves a top F1 score of 0.47. We also evaluated transformer-based models that operate directly on source code. Both CodeBERT and GraphCodeBERT achieved moderate F1 scores (0.23 and 0.34, respectively), suggesting that while transformer-based models can learn from raw code representations, they still fall short of outperforming the best traditional configuration. These results do not reveal any clear relationship between recurrent source code patterns in methods and the likelihood of being performance tested, thus suggesting a lack of generalizable source code characteristics to guide performance testing.

5 Implications

This section discusses some implications of this study along with some directions for future work.

For Practitioners This study gives clear evidence that a significant portion of the codebase of many software systems lacks performance assessment. As software evolves, these code areas may become vulnerable to performance bugs that remain undetected until released. One potential reason for this oversight could be the limited availability of tools for performance test coverage. Indeed, we are unaware of any tool that measures performance test coverage as seamlessly as tools like JaCoCo does for functional testing. We believe that the introduction of such a tool could allow developers to more regularly assess the coverage of their performance testing suites, thus increasing their awareness of the code area that remains unmonitored for performance.

Our results also reveal that distinct performance tests often cover the same code components (i.e., high overlap), even though a significant portion of the codebase remains uncovered by performance test. Although developers might deliberately target the same code components with multiple performance tests to assess their behaviour under varying workloads, this highlights an opportunity to broaden test coverage without incurring additional time costs. We hypothesize that, by raising awareness about performance test coverage, developers might be more inclined to prioritize the creation of tests that target code components currently unassessed for performance. We encourage future work to make it easier for practitioners to be aware of the performance testing suites coverage.

A potential step forward for practitioners is the development of a lightweight *performance-coverage* plugin, similar to functional testing tools like JaCoCo. Based on existing static and dynamic analysis artifacts, such as static code features and method coverage information, this plugin could highlight *coverage hotspots*, i.e., methods with high overlap ratios, and *coverage hotspots*, referring to methods that remain untested for performance. These insights would allow developers to identify redundancies, uncover neglected areas, and better distribute their testing efforts throughout the codebase.

For Researchers Prior work suggests that the high costs of test development and maintainability often hamper the adoption of performance testing in practice (Jangali et al. 2023; Traini 2022; Leitner and Bezemer 2017). In response to this issue, researchers have introduced techniques capable of automatically transforming functional testing suites into performance tests (Jangali et al. 2023). In light of our results, we can formulate educated guesses regarding the potential benefits of these techniques, as well as the challenges that might originate from their adoption. For instance, our results suggest that, by utilizing automated performance test generation, there could be a significant improvement in terms of performance test coverage. Indeed, functional testing suites exhibit significantly higher coverage than that of performance tests (10.4% *vs* 41.3% on average in our dataset), and generated performance tests would inherit such high coverage. However, these benefits might come at a cost, particularly regarding execution time. The high coverage of functional test suites is typically a consequence of their extensive sizes, which may not be feasible for a performance testing suite. In fact, performance tests are typically more time-consuming than functional counterparts (on average 2,507 times more in our dataset). For instance, by using the configuration defined by Jangali et al. (2023) and a typical number of five forks (Traini et al. 2022; Laaber et al. 2020), the time cost of an individual generated performance test would amount to about 400 seconds. For a medium-sized testing suite like logbook, which comprises 564 JUnit tests, this translates to a total time cost of roughly two and a half days. This is approximately 141 times longer than the actual logbook performance testing suite. Even when considering the smallest functional testing suite in our study, namely objenesis, this results in a 5-hour execution time, i.e., 55 times the one of the current performance testing suite. These findings highlight that automated generation alone might not be sufficient to produce performance test suites that are practically usable, given that developers might be deterred by such a time-consuming test process. This underscores a significant challenge for the research community, i.e., automated performance test generation should take into consideration the associated time cost of the generated testing suite.

A potential research direction we aimed to trigger is to devise "smart" selection strategies that automatically identify which code components should undergo performance testing. However, our results indicate that developing a selection strategy aligned with developers' perceptions of what should be tested can be challenging. Our findings highlight a lack of recurrent patterns in source code subject to performance testing, thus suggesting that performance testing should be driven primarily by domain-specific objectives rather than generalizable source code features across software systems. Consequently, researchers should consider other viable alternatives. One option could be to employ test selection strategies that aim to maximize code coverage while mitigating the time cost, for instance, by reducing the number of redundant performance tests covering the same code component (Yoo et al. 2012; Laaber et al. 2024). Another option could be leveraging the workload of software in production, e.g., through execution traces and logs, to understand which code components of the system are more heavily invoked and therefore claim for higher priority in performance testing.

Performance testing inherently addresses dynamic aspects of software, and our work specifically aims to reinforce this perspective by demonstrating that, despite considering a large set of static code features, predicting the code sections that developers deem worthy of performance testing remains unsuccessful. Integrating dynamic profiling tools would conflict with the fundamental premise of this paper, which explicitly restricts the analysis to static features only. It remains obvious that dynamic aspects may change the problem boundaries as well as the model results.

6 Threats to Validity

Construct Validity We restricted the coverage analysis to JMH microbenchmarks and JUnit tests since they are mature and widely adopted frameworks for developing performance and functional testing, respectively. Moreover, both these frameworks operate at the fine-grained level, as they are both used to test individual methods within a codebase. This commonality provides a fair basis for comparing their test coverage.

Another limitation lies in the exclusive use of JMH benchmarks as our performance testing dataset. While JMH is the de facto standard for Java microbenchmarking, its coverage characteristics may differ from those of other performance testing suites, such as load testing frameworks. Obviously, load tests are more effective on more complex computing systems, such as distributed systems, where load can trigger different components with different intensities. In our case, where we basically target execution times of Java methods, load testing effect would be minimal. Nonetheless, as part of our future work, we plan to expand the scope of the study by incorporating additional types of performance testing suites, including tools like JMeter and Gatling, which are commonly used for load testing, possibly applied on more appropriate systems under test.

Using method-level coverage may have limitations, since this metric does not account for cases where performance/functional tests only partially cover the method statements. Our study results may change when employing a statement-level coverage metric. We chose to use method-level coverage due to the significant technical difficulties we encountered when attempting to integrate JMH with conventional statement-level coverage tools. In particular, during the early stages of our study, we experimented with JaCoCo (i.e., one of

the most widely used tools for statement-level coverage analysis in Java) to generate coverage reports from JMH microbenchmarks. However, we were unable to obtain any coverage reports. We suspect that this issue stems from interference between JMH and JaCoCo, as both perform code manipulation in different ways. Specifically, JMH generates auxiliary benchmark classes from the original performance test code to support accurate benchmarking, while JaCoCo inserts probe instructions into every loaded class to track statement execution. We hypothesize that these distinct forms of code manipulation may conflict with each other, potentially explaining the failure to produce coverage reports. Given these challenges, we deemed method-level coverage a reasonable compromise between the practicality of the study and the representativeness of the results. Moreover, method-level coverage has been extensively used in software performance research (Chen et al. 2022, 2023; Traini et al. 2021).

One potential use case for the developed machine learning models is automatically recommending methods to test. If accurate, model predictions would suggest methods typically targeted by performance tests based on their static features. Therefore, we chose performance test coverage as a potential proxy for recommending methods to test, since coverage indicates an explicit intent by developers or performance engineers to evaluate specific execution paths, reflecting some degree of concern about the performance of those methods. This approach also ensures scalability for generating our training/testing datasets, which would otherwise be impractical with approaches like manual expert annotations. However, in some cases even accurate predictions can prevent from identifying performance-critical methods in practice, as performance relevance depends on contextual factors such as expected workload, deployment environment, and practical software usage patterns.

To investigate the feasibility of learning-based approaches for the prediction of performance tests, we fine-tuned two transformer-based models, CodeBERT and GraphCodeBERT, on our dataset. While these models operate directly on source code and eliminate the need for manual feature engineering, they still rely on large-scale pretraining and sufficient fine-tuning data to generalize effectively. Their black-box nature may also limit interpretability compared to traditional ML models based on engineered features. However, we used widely adopted models and followed a well-established fine-tuning pipeline from CodeXGLUE (Lu et al. 2021), making our setup consistent with the current literature in software engineering and code modeling. We did not exhaustively perform hyperparameter tuning across all models to reduce the computational cost of our experiments. Hence, the prediction ability of some models might vary when hyper-parameter tuning is applied.

External Validity We focused solely on *Java software systems*. Our results may not generalize to systems developed in other programming languages. Nevertheless, Java is still among the most used programming languages.[9] The presented analysis is limited to 28 open-source software systems. The findings may not be broadly generalizable; nonetheless, the selected systems are all well-known Java systems encompassing different domains (e.g., database systems, logging frameworks, and web servers). This limited number of subject systems is also motivated by an effort-intensive data collection, which required months of work (in multiple iterations) to get reliable results. This is a known issue in performance engineering that typically restricts the number of subject systems in empirical studies. Nevertheless, the number of subject systems used in our study is larger than most of the recent empirical stud-

[9] Stack Overflow Developer Survey, https://survey.stackoverflow.co/2023.

ies on performance (e.g., see Laaber and Leitner 2018; Laaber et al. 2020; Ding et al. 2020; Chen et al. 2022; Jangali et al. 2023; Zhao et al. 2023).

The coverage labels (*ground truth*) used to train our models are derived from the performance test suites available in the studied projects. As such, the conclusions of this study are influenced by the quality and completeness of these test cases. For example, adding more benchmarks to test suites could affect coverage labels and potentially change the model outcomes. While using real-world test suites improves practical relevance, it also means that any gap or bias in the test cases may influence the results.

The evaluation of transformer-based models was limited to two encoder-only models, CodeBERT and GraphCodeBERT, fine-tuned on a binary classification task using our dataset. Although these models have been widely used in source code modeling tasks, we did not explore larger models (e.g., CodeT5, GPT-based models) or alternative architectures (e.g., graph neural networks or encoder-decoder models). As such, our findings may not generalize to other LLMs or to models optimized for different input encodings (e.g., ASTs, bytecode, or control-flow graphs).

Internal Validity We used a sampling-based CPU profiler to identify the methods covered by tests. These profilers operate by periodically capturing a program's call stack during its execution. A limitation of this approach is the potential omission of call stack information. Since sampling is done at discrete intervals, short-lived function calls or those that fall between sampling points might not be captured. To mitigate this threat, we used async-profiler, which (to our knowledge) provides the lowest sampling rate for profiling Java software.

We employed various pre-processing techniques, including feature selection and class rebalancing, followed by hyperparameter tuning. However, other combinations of data pre-processing methods, algorithms, and hyperparameters could yield better results. Additionally, the source code features used for classifying methods might only partially capture the proper suitability of a method for performance testing. Nevertheless, we have used widely recognized source code features, which have been effectively utilized in previous software performance studies (Laaber et al. 2021). It is also important to note that, due to the inherent variability in machine learning model training, the results may vary slightly if the experiments are repeated.

7 Related Work

Performance Testing The study most closely related to our work is that of Laaber and Leitner (2018), which proposes a performance test quality metric inspired by mutation testing score, namely API benchmarking score (ABS). ABS is related to the concept of test coverage, as it represents the capability of the performance testing suite to find slowdowns. While Laaber and Leitner focus on defining a novel metric for test quality, our research evaluates the quality of existing performance testing suites using traditional code coverage metrics. Traini et al. (2021) show that code components covered by performance tests tend to be less susceptible to refactoring. The execution time cost of performance testing is also related

to this work. Researchers proposed approaches to reduce the time of performance testing without sacrificing results quality (Laaber et al. 2020; Alghmadi et al. 2016; He et al. 2019b; Kalibera and Jones 2013). Lemieux et al. (2018) proposed PerfFuzz, an approach for automatically generating inputs that expose performance issues. Zhang et al. (2011) proposed a mixed symbolic execution approach to generate load testing suites that induce higher system response times and increased memory consumption. Jangali et al. (2023) proposed ju2jmh, a tool that automatically transforms JUnit tests into JMH microbenchmarks.

Test Coverage In Ivanković et al. (2019), the authors describe Google's code coverage infrastructure and how the computed information is visualized and used. The study demonstrates that most of the projects contain few unit tests, despite the opposite perception of the developers. The authors in Zhai et al., 2019) analyzed test coverage data on several widely used Python projects. The main finding is that the coverage strongly depends on the control flow structure. Moreover, the authors found that error-handling code is also neglected. In Gopinath et al. (201), the authors examine the question of coverage criteria as suite quality predictors from the perspective of non-researcher audience. Alves and Visser (2009) conceived an approach for estimating code coverage through static analysis, particularly slicing call graphs.

Performance Bugs Jin et al. (2012b) empirically studied 110 real-world performance bugs collected from 5 open-source software repositories. A more extensive study was recently conducted by Zhao et al. (2023), which investigated 570 performance issues from 13 open-source projects. Other empirical studies have focused on more specific domains, such as internet browsers (Zaman et al. 2012), mobile applications (Mazuera-Rozo et al. 2020), and JavaScript applications (Selakovic and Pradel 2016). While our empirical findings may not directly correlate with the ability to uncover performance issues, there are strong indications of the relevance of code coverage for the efficacy of performance testing. Bach et al. (2017) found that source code covered by functional/performance tests is less prone to bugs. Ding et al. (2020) showed that the code coverage of tests (i.e., scope) influences their capability to uncover performance bugs. The study of Jangali et al. (2023) suggests that the extension of code coverage in performance testing improves the capability of detecting performance issues. These works indicate that incorporating code coverage analysis into performance testing can be beneficial for software performance assurance.

Machine Learning for Performance Testing Laaber et al. (2021) studied the prediction of unstable software benchmarks using static source code features. They focused on benchmarks in open-source projects using the Go programming language. The study demonstrates the effectiveness of combining multiple static source code features and utilizing machine learning models to predict benchmark stability. Chen et al. (2022) presented an approach to build prediction models to detect the tests that are prone to performance regression. Their work focuses on early identifying performance issues to mitigate the performance regressions introduced by new code commits at the test level. They exploit the historical commit information to extract features. Machine learning models are employed to predict performance regressions immediately after a new commit is introduced. The study of Zhao et al. (2024)) proposes an approach for enhancing performance bug prediction using performance code metrics. The study highlights the effectiveness of combining various performance

code metrics with machine learning techniques to identify performance issues. They integrate code characteristics of performance bugs with traditional source code metrics to build machine learning models for predicting presence of performance bugs in the source code.

8 Conclusion and Future Work

This paper presented a comprehensive empirical study focused on performance testing coverage. Our findings revealed the limited coverage of current performance testing suites and the significant execution time cost associated with them. The results of this work suggest opportunities to enhance the coverage of performance testing suites, by emphasizing the necessity to enlighten practitioners about these prevalent limitations.

We have intentionally considered in this paper the concept of code coverage that usually relates to functional testing. Additional metrics should be considered for a sharper concept of performance test coverage, like workload and operational profile. However, these metrics are quite difficult to collect and may sensibly vary for the same application in different contexts. Therefore, we have intended to explore the extent at which performance testing can be solely based on code coverage. Additionally, we have investigated the relationship between source code features and the likelihood of performance testing opportunity. Our goal was to identify recurring characteristics in the source code that could drive performance testing efforts.

As suggested by our findings, the real-world adoption of performance testing techniques might be hampered by their substantial execution time costs. The evidence provided in this paper sustains the idea that the limited coverage of performance tests (as compared to functional ones) only stems from technical issues (e.g., time limits, problems to collect dynamic metrics). Indeed, a wider coverage is desirable as it would allow to identify performance issues in code sections that, for the above reasons, are usually not considered.

To address this challenge, researchers can leverage the automated generation of performance tests (Jangali et al. 2023; Rodriguez-Cancio et al. 2016). One promising direction is the development of "smart" test selection strategies that can reduce the execution time of a performance testing suite without compromising its effectiveness. The selection of code components for performance testing appears to be driven primarily by objectives that cannot be inferred solely from source code features. Instead, these objectives may be closely tied to domain-specific reasons or associated with the operational profile of the software system. Therefore, future performance test generation techniques should better consider these aspects rather than focusing solely on source code features. We encourage future research efforts directed to address this challenge.

An important direction for future work concerns the granularity of coverage analysis. Our study used method-level coverage due to integration limitations between JMH and statement-level tools such as JaCoCo. However, this may overestimate the actual coverage, as entire methods are marked as covered even if only a subset of their statements are executed. Future work should explore instrumentation strategies that enable statement-level coverage without interfering with JMH execution. This may involve investigating more advanced coverage tools, such as OpenClover, or alternatively, developing custom coverage solutions tailored to the specific characteristics of JMH microbenchmarks. For example, this could include designing mechanisms to inject probes at the bytecode level in a manner that avoids disrupting JMH benchmarking integrity.

Another direction for improving the results is to employ alternative approaches such as deep learning. While our evaluation included two transformer based models, CodeBERT and GraphCodeBERT, these models were originally developed for "semantic" tasks, such as code search and summarization, rather than performance-related tasks. Although they provide a useful baseline by leveraging pretrained code representations, they did not outperform traditional ML classifiers in our setting. Future work may explore alternative deep learning models that are more explicitly designed for classification tasks, which could improve performance in identifying the methods to test.

Author Contributions Muhammad Imran: Methodology, data curation, data analysis, and writing of the original draft. Vittorio Cortellessa: Supervision, reviewing, and editing the manuscript. Davide Di Ruscio: Supervision, reviewing, and editing the manuscript. Riccardo Rubei: Machine Learning, data analysis, and editing the manuscript. Luca Traini: Conceptualization, reviewing and editing the manuscript.

Funding Open access funding provided by Università degli Studi dell'Aquila within the CRUI-CARE Agreement. This work is partially supported by Italian Government (Ministero dell'Università e della Ricerca, PRIN 2022 PNRR): "RECHARGE: monitoRing, tEsting, and CHaracterization of performAnce Regressions" (cod.P2022SELA7), and by "ICSC– Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing," funded by European Union– NextGenerationEU.

Data Availability The dataset and source code used in this study are publicly available online at the following repository: Replication Package.

Declarations

Conflicts of Interests The authors declare no conflict of interest related to this manuscript.

References

Alam MMU, Liu T, Zeng G, Muzahid A (2017) Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In: Proceedings of the Twelfth European conference on computer systems. pp 298–313

Alcocer JPS, Bergel A (2015) Tracking down performance variation against source code evolution. ACM SIGPLAN Notices 51(2):129–139

Alcocer JPS, Bergel A, Valente MT (2020) Prioritizing versions for performance regression testing: the pharo case. Sci Comput Program 191:102415

Alghmadi HM, Syer MD, Shang W, Hassan AE (2016) An automated approach for recommending when to stop performance tests. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp 279–289. https://doi.org/10.1109/ICSME.2016.46

Alsaqaf W, Daneva M, Wieringa R (2019) Quality requirements challenges in the context of large-scale distributed agile: an empirical study. Inf Softw Technol 110:39–55

Alves TL, Visser J (2009) Static estimation of test coverage. In: 2009 Ninth IEEE international working conference on source code analysis and manipulation. IEEE, pp 55–64

Avinash M, Nithya M, Aravind S (2022) Automated machine learning-algorithm selection with fine-tuned parameters. In: 2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS). IEEE, pp 1175–1180

Bach T, Andrzejak A, Pannemans R, Lo D (2017) The impact of coverage on bug density in a large industrial software project. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp 307–313

Barrett E, Bolz-Tereick CF, Killick R, Mount S, Tratt L (2017) Virtual machine warmup blows hot and cold. Proc ACM Program Lang 1(OOPSLA)

Batista GE, Prati RC, Monard MC (2004) A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explorations Newsl 6(1):20–29

Behutiye W, Karhapää P, López L, Burgués X, Martínez-Fernández S, Vollmer AM, Rodríguez P, Franch X, Oivo M (2020) Management of quality requirements in agile and rapid software development: a systematic mapping study. Inf Softw Technol 123:106225

Brodersen KH, Ong CS, Stephan KE, Buhmann JM (2010) The balanced accuracy and its posterior distribution. In: 2010 20th international conference on pattern recognition. pp 3121–3124. https://doi.org/10.1109/ICPR.2010.764

Brown S, Mitchell A, Power JF (2005) A coverage analysis of Java benchmark suites

Brutlag J (2009) Google AI blog: speed matters. https://ai.googleblog.com/2009/06/speed-matters.html

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357

Chen J, Ding Z, Tang Y, Sayagh M, Li H, Adams B, Shang W (2023) IoPV: On inconsistent option performance variations. In: Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2023. ACM, pp 845—857. https://doi.org/10.1145/3611643.3616319

Chen J, Shang W, Shihab E (2022) PerfJIT: test-level just-in-time prediction for performance regression introducing commits. IEEE Trans Software Eng 48(5):1529–1544

Chung L, Nixon BA, Yu E, Mylopoulos J (2012) Non-functional requirements in software engineering, vol 5. Springer Science & Business Media

Collard ML, Decker MJ, Maletic JI (2011) Lightweight transformation and fact extraction with the srcML toolkit. In: 2011 IEEE 11th international working conference on source code analysis and manipulation. pp 173–184. https://doi.org/10.1109/SCAM.2011.19

Corporation O (2016) Java Microbenchmarking Harness (JMH). https://openjdk.org/projects/code-tools/jmh/

Costa D, Andrzejak A, Seboek J, Lo D (2017) Empirical study of usage and performance of java collections. In: Proceedings of the 8th ACM/SPEC on international conference on performance engineering. pp 389–400

Costa D, Bezemer CP, Leitner P, Andrzejak A (2019) What's wrong with my benchmark results? studying bad practices in JMH benchmarks. IEEE Trans Software Eng 47(7):1452–1467

De Oliveira AB, Fischmeister S, Diwan A, Hauswirth M, Sweeney PF (2017) Perphecy: performance regression test selection made simple but effective. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp 103–113. https://doi.org/10.1109/ICST.2017.17

Ding Z, Chen J, Shang W (2020) Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet? In: Rothermel G, Bae D (eds) ICSE '20: 42nd international conference on software engineering, Seoul, South Korea, 27 June - 19 July, 2020. ACM, pp 1435–1446. https://doi.org/10.1145/3377811.3380351

Eadline D (2005) Micro-benchmarks vs macro-benchmarks. https://www.clustermonkey.net/Benchmarking-Methods/micro-benchmarks-vs-macro-benchmarks.html. Accessed Jul 2024

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al (2020) CodeBERT: a pre-trained model for programming and natural languages. arXiv:2002.08155

Fu MQ, Wei M, Qiao M, Ji P, Deng Z, Cui D, Zhao Y (2024) HGNN4Perf: detecting performance optimization opportunities via hypergraph neural network. In: Proceedings of the 15th Asia-pacific symposium on internetware. pp 279–282

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07. ACM, pp 57–76. https://doi.org/10.1145/1297027.1297033

Google (2021) Google Caliper. https://github.com/google/caliper

Gopinath R, Jensen C, Groce A (2014) Code coverage for suite evaluation by developers. In: Jalote P, Briand LC, van der Hoek A (eds) 36th international conference on software engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. ACM, pp 72–82. https://doi.org/10.1145/2568225.2568278

Grambow M, Kovalev D, Laaber C, Leitner P, Bermbach D (2022) Using microbenchmark suites to detect application performance changes. https://doi.org/10.48550/ARXIV.2212.09515

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, et al (2020) GraphCodeBERT: pre-training code representations with data flow. arXiv:2009.08366

Guyon I, Weston J, Barnhill S, Vapnik V (2002) Gene selection for cancer classification using support vector machines. Mach Learn 46:389–422

Häubl C, Wimmer C, Mössenböck H (2013) Deriving code coverage information from profiling data recorded for a trace-based just-in-time compiler. In: Proceedings of the 2013 international conference on principles and practices of programming on the Java platform: virtual machines, languages, and tools. pp 1–12

He S, Liu T, Lama P, Lee J, Kim IK, Wang W (2021) Performance testing for cloud computing with dependent data bootstrapping. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 666–678

He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (2019a) A statistics-based performance testing methodology for cloud applications. In: Proceedings of the 2019 27th ACM Joint Meeting on European software engineering conference and symposium on the foundations of software engineering. pp 188–199

He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (2019b) A statistics-based performance testing methodology for cloud applications. In: Proceedings of the 2019 27th ACM Joint Meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019. ACM, pp 188–199

Huang P, Ma X, Shen D, Zhou Y (2014) Performance regression testing target prioritization via performance risk analysis. In: Proceedings of the 36th international conference on software engineering. pp 60–71

Imran M, Cortellessa V, Di Ruscio D, Rubei R, Traini L (2024) An empirical study on code coverage of performance testing. In: Proceedings of the 28th international conference on evaluation and assessment in software engineering, EASE '24. Association for Computing Machinery, New York, NY, USA, pp 48–57. https://doi.org/10.1145/3661167.3661196

Imran M, Cortellessa V, Di Ruscio D, Rubei R, Traini L (2024) Is code coverage of performance tests related to source code features? An empirical study on open-source systems - replication package. https://github.com/SpencerLabAQ/performance-test-coverage

Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, pp 435–445

ISO/IEC: ISO/IEC 25010 - System and software quality models. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Retrieved July 2024

Ivanković M, Petrović G, Just R, Fraser G (2019) Code coverage at google. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019. ACM, pp 955–963

Jangali M, Tang Y, Alexandersson N, Leitner P, Yang J, Shang W (2023) Automated generation and evaluation of JMH microbenchmark suites from unit tests. IEEE Trans Software Eng 49(4):1704–1725. https://doi.org/10.1109/TSE.2022.3188005

Jiang ZM, Hassan AE (2015) A survey on load testing of large-scale software systems. IEEE Trans Software Eng 41(11):1091–1118

Jiarpakdee J, Tantithamthavorn C, Treude C (2018) Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE Computer Society, pp. 92–103

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices 47(6):77–88

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012b) Understanding and detecting real-world performance bugs. In: Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, PLDI '12. ACM, pp. 77–88. https://doi.org/10.1145/2254064.2254075

Kalibera T, Jones R (2013) Rigorous benchmarking in reasonable time. In: Proceedings of the 2013 international symposium on memory management, ISMM '13. ACM, pp 63–74. https://doi.org/10.1145/2491894.2464160

Kalibera T, Jones R (2020) Quantifying performance changes with effect size confidence intervals. arXiv:2007.10899

Laaber C, Basmaci M, Salza P (2021) Predicting unstable software benchmarks using static source code features. Empir Softw Eng 26(6):114. https://doi.org/10.1007/s10664-021-09996-y

Laaber C, Gall HC, Leitner P (2021) Applying test case prioritization to software microbenchmarks. Empir Softw Eng 26(6):133. https://doi.org/10.1007/s10664-021-10037-x

Laaber C, Leitner P (2018) An evaluation of open-source software microbenchmark suites for continuous performance assessment. ACM

Laaber C, Würsten S, Gall HC, Leitner P (2020) Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2020. ACM, pp 989–1001

Laaber C, Yue T, Ali S (2024) Evaluating search-based software microbenchmark prioritization. IEEE Transactions on Software Engineering. pp 1–16. https://doi.org/10.1109/TSE.2024.3380836
Leitner P, Bezemer CP (2017) 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, ICPE '17. ACM, pp 373–384. https://doi.org/10.1145/3030207.3030213
Lemieux C, Padhye R, Sen K, Song D (2018) Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018. ACM, pp 254–265. https://doi.org/10.1145/3213846.3213874
Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, et al (2021) Codexglue: a machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664
Maricq A, Duplyakin D, Jimenez I, Maltzahn C, Stutsman R, Ricci R (2018) Taming performance variability. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA, pp 409–425
Mazuera-Rozo A, Trubiani C, Linares-Vásquez M, Bavota G (2020) Investigating types and survivability of performance bugs in mobile apps. Empir Softw Eng 25:1644–1686
McCabe TJ (1976) A complexity measure. IEEE Trans Software Eng 4:308–320
Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009) Producing wrong data without doing anything obviously wrong! In: Proceedings of the 14th international conference on architectural support for programming languages and operating systems, ASPLOS XIV. Association for Computing Machinery, New York, NY, USA, pp 265–276. https://doi.org/10.1145/1508244.1508275
Noller Y, Kersten R, Păsăreanu CS (2018) Badger: Complexity analysis with fuzzing and symbolic execution. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018. ACM, pp 322–332. https://doi.org/10.1145/3213846.3213868
O'Connor N (n.d.) JUnitPerf. https://github.com/noconnor/JUnitPerf
Olenski S (2016) Why brands are fighting over milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/
Piwowarski P, Ohba M, Caruso J (1993) Coverage measurement experience during function test. In: Proceedings of the 15th international conference on software engineering, ICSE '93. IEEE Computer Society Press, pp 287–301
Reichelt DG, Kühne S (2018) Better early than never: performance test acceleration by regression test selection. In: Companion of the 2018 ACM/SPEC international conference on performance engineering, ICPE '18. ACM, pp 127–130. https://doi.org/10.1145/3185768.3186289
Rodriguez-Cancio M, Combemale B, Baudry B (2016) Automatic microbenchmark generation to prevent dead code elimination and constant folding. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE '16. ACM, pp 132–143. https://doi.org/10.1145/2970276.2970346
Rohella A, Takada S (2018) Testing android applications using multi-objective evolutionary algorithms with a stopping criteria. In: SEKE, pp 308–307
Samoaa H, Leitner P (2021) An exploratory study of the impact of parameterization on JMH measurement results in open-source projects. In: Proceedings of the ACM/SPEC international conference on performance engineering, ICPE '21. ACM, pp 213–224. https://doi.org/10.1145/3427921.3450243
Selakovic M, Pradel M (2016) Performance issues and optimizations in javascript: an empirical study. In: Proceedings of the 38th international conference on software engineering. pp 61–72
Shipilev A (2014) Nanotrusting nanotime. https://shipilev.net/blog/2014/nanotrusting-nanotime/. Accessed 04 Jul 2024
Smith CU, Williams LG (2002) Performance solutions: a practical guide to creating responsive, scalable software, vol 23. Addison-Wesley Reading
Stefan P, Horky V, Bulej L, Tuma P (2017) Unit testing performance in java projects: Are we there yet? In: Proceedings of the 8th ACM/SPEC on international conference on performance engineering. pp 401–412
Traini L (2022) Exploring performance assurance practices and challenges in agile software development: an ethnographic study. Empir Softw Eng 27(3):74. https://doi.org/10.1007/s10664-021-10069-3
Traini L, Cortellessa V, Di Pompeo D, Tucci M (2022) Towards effective assessment of steady state performance in java software: are we there yet? Empir Softw Eng 28(1):13. https://doi.org/10.1007/s10664-022-10247-x
Traini L, Di Menna F, Cortellessa V (2024) AI-driven java performance testing: balancing result quality with testing time. In: Proceedings of the 39th IEEE/ACM international conference on automated software engineering, ASE '24. Association for Computing Machinery, New York, NY, USA, pp 443–454. https://doi.org/10.1145/3691620.3695017
Traini L, Di Pompeo D, Tucci M, Lin B, Scalabrino S, Bavota G, Lanza M, Oliveto R, Cortellessa V (2021) How software refactoring impacts execution time. ACM Trans Softw Eng Methodol 31(2). https://doi.org/10.1145/3485136

Vokolos FI, Weyuker EJ (1998) Performance testing of software systems. In: Proceedings of the 1st international workshop on software and performance. pp 80–87

Walker A, Coffey M, Tisnovsky P, Cerny T (2020) On limitations of modern static analysis tools. In: Kim KJ, Kim HY (eds) Information science and applications. Springer Singapore, pp 577–586

Weyuker EJ, Vokolos FI (2000) Experience with performance testing of software systems: issues, an approach, and case study. IEEE Trans Software Eng 26(12):1147–1156. https://doi.org/10.1109/32.888628

Yang Q, Li JJ, Weiss D (2006) A survey of coverage based testing tools. In: Proceedings of the 2006 international workshop on Automation of software test. pp 99–103

Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab 22(2):67–120

Zaman S, Adams B, Hassan AE (2012) A qualitative study on performance bugs. In: Proceedings of the 9th IEEE working conference on mining software repositories, MSR '12. IEEE Press, pp 199–208

Zhai H, Casalnuovo C, Devanbu P (2019) Test coverage in python programs. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, pp 116–120

Zhang P, Elbaum S, Dwyer MB (2011) Automatic generation of load tests. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). pp 43–52. https://doi.org/10.1109/ASE.2011.6100093

Zhao G, Georgiou S, Hassan S, Zou Y, Truong D, Corbin T (2024) Enhancing performance bug prediction using performance code metrics. In: Proceedings of the 21st international conference on mining software repositories, MSR '24. Association for Computing Machinery, New York, NY, USA, pp 50–62. https://doi.org/10.1145/3643991.3644920

Zhao Y, Xiao L, Bondi AB, Chen B, Liu Y (2023) A large-scale empirical study of real-life performance issues in open source projects. IEEE Trans Software Eng 49(2):924–946

Zhao Y, Xiao L, Wang X, Sun L, Chen B, Liu Y, Bondi AB (2020) How are performance issues caused and resolved?-an empirical study from a design perspective. In: Proceedings of the ACM/SPEC international conference on performance engineering. pp 181–192

Zhao Y, Xiao L, Xiao W, Chen B, Liu Y (2019) Localized or architectural: an empirical study of performance issues dichotomy. In: 2019 IEEE/ACM 41st international conference on software engineering: companion proceedings (ICSE-Companion). IEEE, pp 316–317

Zhou Z, Zhou Y, Fang C, Chen Z, Tang Y (2022) Selectively combining multiple coverage goals in search-based unit test generation. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering. pp 1–12