Identifying and Replicating Code Patterns Driving Performance Regressions in Software Systems

Denivan Campos* Luana Martins†, Emanuela Guglielmi*, Michele Tucci‡,
Daniele Di Pompeo‡, Simone Scalabrino*, Vittorio Cortellessa‡, Dario Di Nucci†, Rocco Oliveto *
denivan.dasilva@unimol.it, lalmeida.martins@unisa.it, emanuela.guglielmi@unimol.it, michele.tucci@univaq.it,
daniele.dipompeo@univaq.it, simone.scalabrino@unimol.it, vittorio.cortellessa@univaq.it, ddinucci@unisa.it,
rocco.oliveto@unimol.it
*University of Molise, Pesche, Italy
†University of Salerno, Salerno, Italy
‡University of L'Aquila, L'Aquila, Italy

***Abstract*—Context: Performance regressions negatively impact execution time and memory usage of software systems. Nevertheless, there is a lack of systematic methods to evaluate the effectiveness of performance test suites. Performance mutation testing, which introduces intentional defects (mutants) to measure and enhance fault-detection capabilities, is promising but under-explored. A key challenge is understanding if generated mutants accurately reflect real-world performance issues. Goal: This study evaluates and extends mutation operators for performance testing. Its objectives include (i) collecting existing performance mutation operators, (ii) introducing new operators from real-world code changes that impact performance, and (iii) evaluating these operators on real-world systems to see if they effectively degrade performance. Method: To this aim, we will (i) review the literature to identify performance mutation operators, (ii) conduct a mining study to extract patterns of code changes linked to performance regressions, (iii) propose new mutation operators based on these patterns, and (iv) apply and evaluate the operators to assess their effectiveness in exposing performance degradations. Expected Outcomes: We aim to provide an enriched set of mutation operators for performance testing, helping developers and researchers identify harmful coding practices and design better strategies to detect and prevent performance regressions.**

***Index Terms*—Performance Issues, Performance Mutation Testing, Mining Software Repositories, Fault Injection.**

I. Introduction

Software performance issues are non-functional problems that reduce the efficiency, responsiveness, or scalability of software due to factors like slow processing, resource overuse, or system bottlenecks, ultimately harming user experience and wasting resources [1]–[3]. These issues are challenging to detect and resolve because their occurrence often depends on variations in the execution environment, such as hardware configurations, workload patterns, or system settings. Performance issues are usually detected by running performance tests (*i.e.*, *benchmarks*) with specific inputs to check whether performance metrics (*e.g.*, execution time) degrade as the software evolves. However, the lack of reliable testing oracles, *e.g.*, how slow should a computation be to be considered a performance issue, makes it challenging to identify and diagnose these issues consistently [4]–[6].

The literature highlights the critical role of performance testing in mitigating performance degradation and ensuring systems operate without issues [7]–[12]. Despite its importance, there remains a significant gap in systematic methods for evaluating the effectiveness of performance test suites [12], [13].

In this regard, performance mutation testing offers a promising approach. Like functional mutation testing, performance mutation consists of introducing intentional performance issues (*mutants*) to assess the detection capabilities of performance tests. Sánchez et al. [6], [14] and Delgado-Pérez et al. [3] explore the possible benefits and limitations of applying mutation testing for performance and define performance mutation testing (PMT). Like the classic mutation test, PMT aims to generate variants of the original program, *i.e.*, performance mutants, where each variation simulates a performance error. They also identified some challenges related to mutation testing and proposed seven new operators to model known patterns in C/C++ for inducing performance bugs. The authors compared the effectiveness of classical and performance mutants and found that the latter can be used to evaluate and improve the effectiveness of performance testing.

Despite its potential, PMT is still an under-explored field. A key challenge lies in the *limited understanding of whether artificially generated mutants accurately reflect the performance issues in real-world systems*. More specifically, it is still unknown to what extent the generated mutants are relevant in different contexts. This gap limits developers' ability to evaluate the robustness of performance tests in identifying and mitigating actual performance regressions.

This registered report describes a study that evaluates and extends the mutation operators relevant to performance testing. The study begins with a comprehensive literature review to establish a catalog of existing performance mutation operators. The catalog will be enriched starting from the literature by introducing new operators derived analyzing real-world code changes that have demonstrably degraded system performance. To achieve this, we will leverage established benchmarks to identify such changes and extract patterns that will inform the design of novel mutation operators. Finally, we will evaluate the effectiveness of both existing and newly developed mutation operators on real-world systems, assessing their ability to simulate performance degradations. The evaluation

will determine whether current performance tests can reliably detect the slowdowns introduced by these mutants, providing insights into their applicability across various contexts.

Our findings are expected to advance performance mutation testing by providing an enriched set of mutation operators and empirical evidence of their impact on performance testing. The findings can assist in identifying harmful coding practices and design more effective strategies to detect and mitigate performance regressions, thus contributing to developing more robust and reliable performance testing methodologies.

The remainder of this paper is structured as follows. Section II presents the related work. Section III presents the goal of the study and its research questions, which will be answered leveraging the methodology provided in Section IV, and whose main threats to validity are reported in Section V. Finally, Section VI concludes the paper.

II. Related Works

This section describes the related work concerning mutation testing for bug replication and performance mutation testing.

A. Mutation Testing for Bug Replication

Just et al. [15] investigated the correlation between detecting mutants generated using frequently used mutation operators and real defects. In particular, the authors used MAJOR mutation framework provides the following set of mutation operators [16]: *Replace constants*, *Replace operators*, *Modify branch conditions*, and *Delete statements*. Furthermore, they examined whether code coverage could improve the efficacy of mutation analysis, showing a statistically significant correlation between mutant detection and actual failures, regardless of code coverage.

To determine whether classic mutation scores and fault detection efficacy are associated, Papadakis et al. [17] examined the relationship between mutation scores and real fault detection. They analyzed C and JAVA programs featuring real faults to demonstrate how the size of the test set affects the correlations between mutation scores and fault detection. To do so, the authors applied the mutation operators: *Arithmetic, Logical, Conditional, and Relational Operators Replacement*, *Operator Replacement Unary*, *Statement Deletion*, and *Literal Value Replacement*. The results show that achieving higher mutation scores improves fault detection significantly and indicate that mutants provide good guidance for improving fault detection of test suites, although their correlation with fault detection is weak.

The existing performance mutation operators provide useful foundations for fault detection in various types of software testing. In the referred papers, Just et al. [15] and Papadakis et al. [17] examine the relationship between traditional mutation analysis and actual fault detection, emphasizing the effectiveness of existing mutation operators and their correlation with fault detection in functional tests. While these operators are instrumental in verifying functional correctness, they do not directly address performance slowdowns related to suboptimal memory and resource management.

Complementary, Wu et al. [18] introduced memory mutation to replicate memory-related faults. They proposed memory mutation operators (e.g., *Replace calloc with malloc, calloc with alloca, and malloc with alloca*, *Remove null character assignment statement*, *Replace dynamic memory allocation calls*) and compared them to traditional mutation operators. In addition, they addressed the problems associated with equivalent and duplicated mutants. Their results show that traditional operators insufficiently capture memory faults and reduce the effectiveness of test suites by 44%.

While the memory mutation operators proposed by Wu et al. [18] are valuable, they do not comprehensively simulate performance issues that involve memory management, data locality, thread synchronization, and cache usage. In contrast, our study diverges by targeting performance tests, explicitly aiming to expand and refine mutation operators derived from real-world performance problems and evaluating their capability to produce realistic performance degradations.

B. Performance Mutation Testing

Delgado-Pérez et al. [3] investigated the application of performance mutation testing (PMT) to improve the detection of software performance bugs. They proposed seven performance mutation operators related to execution time (e.g., *Loop perturbation*, *Method call*, *Conditional statement*) and memory consumption (e.g., *Object generation*, *Collections*). The authors then compared the effectiveness of performance mutation operations with traditional ones. The results show that performance mutants point out performance degradation while preserving the semantics of the original program to improve performance test effectiveness.

Jangali et al. [19] performed a comparison on the effectiveness of manually-written microbenchmarks with microbenchmarks generated with JU2JMH and other state-of-the-art tools in detecting real performance bugs. In detail, it features five performance mutant operators (i.e., *Primitive to Wrapper*, *StringBuilder to StringBuffer*, *StringBuilder to StringBuffer*, *Swap of Operands in Condition*, and *Simulation of Heavy-Weight Operations*) to evaluate the quality of microbenchmarks during performance bug detection. The results show that the JU2JMH benchmarks can cover more of the software applications than manually-written benchmarks.

Chen et al. [20] investigated whether synthetic bugs can be used to evaluate performance bug diagnosis tools, improve the quality of performance testing methodologies, and identify areas that need improvement in performance bug detection and localization approaches. They proposed a framework that leverages PMT to simulate software performance bugs and identify fault detection in C\C++. Their performance operators (e.g., *replace dfaisfast and fgrep calls with 0*, *remove cache memoization*, *remove early break from loops*, and *prepend 1* loop to loop bodies*) were derived from real performance bugs and software optimizations, analyzing existing cases in the literature and real-world software projects.

These related studies [3], [19], [20] have introduced operators targeting specific aspects like execution time and memory

consumption. However, they may not fully capture the wide variety of performance issues encountered in real-world applications. For example, a recent literature review [21] identifies three primary categories of performance bugs related to time, memory, and energy consumption. While the related works present operators for execution time and memory consumption, operators targeting energy consumption are still lacking. In addition, real-world performance bugs can arise from other factors besides data structures, such as redundant operations, misused algorithms, or inappropriate resource handling. Therefore, our study aims to collect existing performance mutation operators, propose new ones from the real world, and evaluate their effectiveness in identifying performance degradation.

C. Our paper

Below, we highlight the key aspects that will set our study apart and enhance its insights.

Mining real-world performance issues. Instead of relying on known causes of performance degradation from the literature, we will mine performance related issues from open-source projects. Specifically, we will manually analyze pairs of code before and after fixes to identify new performance mutation operators. We will work on performance issues confirmed as significant by developers from the respective projects, ensuring that the mutation operators we will define reflect real-world concerns rather than theoretical assumptions.

Validation through benchmarking. We will run benchmarks on the commits before and after the fix to (i) further validate whether the problem was indeed a performance issue and (ii) confirm that the applied fixes improve performance. Our study brings a practical perspective on what developers consider important to fix as performance issues, offering insights on how to fix it based on real-world development practices.

Different research goals. The referenced paper compares performance mutants against traditional mutants to evaluate their ability to induce perceptible performance degradation. In contrast, our goal is to identify and define new performance mutation operators that reflect real-world performance issues. We focus on discovering and systematizing these operators rather than comparing their impact against traditional mutants.

Different programming language. The mutation operators discussed in related works, which simulate issues such as memory consumption, workload imbalances, or unnecessary operations, have primarily been applied to C/C++ programs. However, performance characteristics can vary significantly across programming languages. Therefore, we will consider these distinct characteristics and better simulate real-world performance issues in Java applications.

Broader Coverage of Performance Issues. Unlike traditional functional testing [15], [17], [18], performance testing requires different considerations and approaches. Previous works have proposed performance mutation operators focusing on execution time and memory consumption. We aim to propose new mutation operators that account for broader performance concerns (e.g., energy consumption), reflecting a more comprehensive view of real-world performance bugs.

III. Research Questions and Objectives

The *goal* of our empirical study is to define performance mutation operators based on real-world coding patterns and evaluate their effectiveness to inject realistic slowdowns across different contexts, i.e., specific code structure. The *purpose* of the study will be to (i) perform a literature review to identify a comprehensive set of performance mutation operators, (ii) extract coding patterns associated with performance slowdowns in real-world project, (iii) propose new performance mutant operators to stimulate the coding practices leading to performance slowdown, and (iv) identify whether the performance mutant operators can generate slowdowns in different contexts. By exploring the alignment between artificial mutants and genuine performance issues, this study will enhance our ability to evaluate and improve performance testing strategies and provide actionable insights for anticipating and mitigating performance regressions. Thus, the *perspective* will be of both practitioners and researchers interested in understanding the coding patterns related to real-world performance issues to have practical insights and enhance their debugging and optimization workflows for detecting and resolving performance problems. Our empirical investigation follows a sequential approach, addressing the following research questions (**RQ**):

RQ$_0$. *Do developers use benchmarking to confirm performance issues and their resolution?*

The motivation behind **RQ$_0$** is twofold. First, we aim to examine how frequently developers rely on benchmarking to detect and validate performance issues. Second, when benchmarks are available, we will analyze whether their outcomes confirm the effectiveness of the applied fixes in resolving the identified issues. A lack of benchmarking for performance bugs or benchmarks that fail to demonstrate meaningful improvements may highlight critical gaps in current practices. In this case, we will generate the benchmarks to confirm the effectiveness of the applied fixes. Then, we will deepen our analysis by exploring the following research questions:

RQ$_1$. *What are the coding patterns related to performance issues in real-world software systems?*

RQ$_2$. *What performance mutation operators can simulate performance issues based on the identified coding practices?*

In **RQ$_1$**, we aim to uncover the coding patterns and root causes contributing to real-world software performance issues. We will construct a structured taxonomy that links specific coding practices to performance degradations by analyzing these patterns, providing a foundational understanding of how performance issues arise. Building on insights of **RQ$_1$**, with **RQ$_2$** we will focus on deriving performance mutation operators replicating the coding changes leading to performance

issues. The identified set of mutation operators will enrich the existing catalog of performance mutants from literature.

RQ$_3$. *To what extent are performance mutants valid substitutes for real-world performance issues in software performance testing?*

In **RQ$_3$**, we will validate the entire catalog of performance mutation operators—both those identified in the literature and the newly introduced ones. This evaluation will assess their generalizability across different contexts. Please, notice that we consider "context" as the specific code structure where a mutation operator is introduced. Since not all mutants can be applied in all contexts (e.g., a mutant that modifies a predicate clause requires a conditional expression), our methodology accounts for these constraints. To refine this investigation, we pose two sub-questions:

RQ$_{3.1}$. *To what extent do performance mutants differ in induced slowdowns?*

RQ$_{3.2}$. *What is the influence of the injection context on the effectiveness of performance mutants?*

In **RQ$_{3.1}$**, we will evaluate the magnitude of slowdowns introduced by different mutation operators and their detectability within the same code structure (if applicable) to analyze their consistency and impact. Meanwhile, **RQ$_{3.2}$** will extend this analysis to assess the consistency and variability of each mutation operator the same mutant operators across different code structures (if applicable) to determine its effectiveness and impact in varying contexts.

We aim to advance the understanding of performance mutation testing by systematically addressing these research questions. We offer practical tools and insights to improve performance testing practices, anticipate performance issues, and validate comprehensive catalogs of performance mutation operators in real-world software systems.

IV. Methodology

To design and report our empirical study, we adhere to the empirical software engineering guidelines outlined by Wohlin *et al.* [22] and the ACM/SIGSOFT Empirical Standards[1], particularly the standards for "General Standard", "Repository Mining", and "Benchmarking". Figure 1 provides an overview of our three-step study design, which we detail below.

A. Identifying Coding Patterns Leading to Performance Bugs

Searching the literature on performance issues. We conducted a preliminary literature review (discussed in Section II) to identify: (i) An initial set of performance mutation operators (Table I); (ii) Curated datasets of performance-related issues. We consolidated data from three manually curated datasets [10], [23], [24], resulting in 497 performance-related issues extracted from Apache Software Foundation projects. This merged dataset forms our *golden standard dataset*, which will serve as a starting point for analyzing and extending the catalog of performance mutation operators.

Tracing bug-fixing commits and microbenchmarks. The golden standard dataset includes bug descriptions but lacks corresponding code or benchmarks. To address this, we will trace each issue to its bug-fixing commit and associated benchmarks using PyDriller [25]. We will identify benchmarks targeting the affected code by mining the modified files and methods in each commit. Microbenchmarking is a performance testing technique that evaluates the efficiency of small, isolated code snippets. It operates at a fine-grained level, similar to unit testing, and helps assess the impact of specific code changes on execution performance. Using microbenchmarking will help us confirm whether the code altered to fix a performance issue is actually to solve it [26]. Issues without identifiable bug-fixing commits or benchmarks will be excluded from our analysis.

Executing microbenchmarks. For issues with valid bug-fixing commits and benchmarks, we will execute the benchmarks on both the bug-fixing and predecessor commits to collect performance metrics (i.e., *execution time*, and *memory usage*). Issues will be excluded if: (i) The benchmarks fail to build or run; (ii) The predecessor commit is unavailable or incompatible with the benchmarks. Please note that if the benchmarks to validate the performance issues in the *Golden Dataset* are not available, we will create the benchmarks with the Ju2JMH, a tool that converts JUnit tests into JMH microbenchmarks [19]. This step will produce a dataset of performance metrics detailing the execution time of benchmarks before and after fixes, enabling analysis for **RQ$_0$**. As for **RQ$_3$**, we will extend the dataset of projects to evaluate the proposed mutation operators in real-world scenarios. Therefore, we will execute the benchmarks in bug-fixing and predecessor commits to collect performance metrics (i.e., *execution time*, *memory usage*, and *Mutation Score*.

Identifying root causes and coding patterns. The code pairs (before and after fixes) associated with performance issues in the dataset will be analyzed to identify root causes and coding patterns. We will use an LLM, GPT-4o, as an assistant to automate the process of identifying potential coding patterns associated with the performance issues. According to MacNiel et al [27], GPT-like LLMs can create explanations for the code, reducing the cognitive demand to understand complex codes from different systems. In addition, according to Colavito et al. [28], GPT-like LLMs demonstrate a high level of agreement with human annotators, which could support us classifying the issues. Therefore, instead of asking GPT4-o to define recurring practices outright, we will have preliminary classifications and explanations, which we will manually validate across multiple instances. In particular, two experts will review and perform an open-coding of the explanation given by the GPT-4o regarding the potential root causes and coding patterns to answer **RQ$_1$**.

[1]Available at: https://github.com/acmsigsoft/EmpiricalStandards

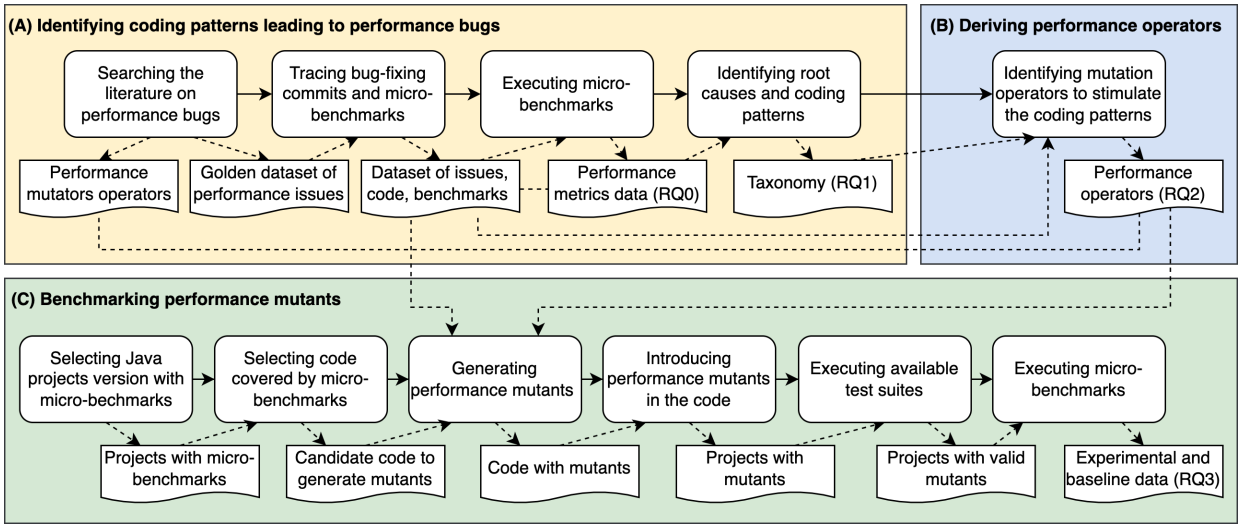

Fig. 1. Overview of the Research Method.

TABLE I
PERFORMANCE MUTATION OPERATORS FROM LITERATURE

Acronym	Operator	Description	Ref.
RCL	Removal of Stop Condition in Loop	RCL removes a stop condition in a loop to keep iterating until another condition is satisfied.	[3]
URV	Unnecessary Recalculation of Values	URV seeks for variables defined to store the value returned by an invocation to a method to force the recalculation of values.	[3]
MSL	Move/Copy Statement into Loop	MSL searches for the generation of objects before a loop statement and moves it into the loop.	[3]
SOC	Swap of Operands in Condition	SOC swaps the operands in a condition linked by a binary logical operator (&& and ||) to evaluate the most time-consuming condition regardless of the other conditions.	[3]
HWO	Simulation of Heavy-Weight Operation	HWO injects a delay right after each invocation to methods in third-party libraries and known heavy-weight operations (storage access, network connection).	[3]
CSO	Creation of Short-lived Objects	CSO targets the methods that receive an object as a parameter and generates a clone of such objects, producing new short-lived objects every time the method is invoked.	[3]
MSR	Memory Space Reservation	MSR modifies a collection with dynamic allocation to shrink or expand the reservation space for elements to simulate both cases.	[3]
PTW	Primitive to Wrapper	PTW replaces a primitive type (e.g., long) with its corresponding wrapper class (i.e., Long).	[19]
STS	StringBuilder to StringBuffer	STS replaces a java.lang.StringBuilder object with a java.lang.StringBuffer.	[19]
EFL	Enhanced For Loops	EFL replaces a traditional for-loop with a for-each loop to iterate over an array or a Collections class.	[19]

B. Deriving Performance Operators

Using the taxonomy of coding patterns from $\mathbf{RQ}_1$, we will employ GPT-4o to suggest mutation operators that simulate performance issues. The rationale behind using fix explanations to inform mutation operators is based on the intuition that fix patterns provide direct insight into recurring code transformations needed to resolve performance issues. It will use the coding patterns identified in the taxonomy to suggest potential code transformations that could degrade performance. For example, if the taxonomy identifies a coding pattern related to the use of collections, then GPT-4o could propose mutation operators that replaces data structures with a less efficient alternative, such as replacing HashMap with a LinkedHashMap. Then, similar to the previous step, two experts will analyze the GPT4-o answers to propose the final set of performance mutation operators. The newly generated operators will be merged with the catalog of performance mutation operators identified in the literature (Table I). This enriched catalog will answer $\mathbf{RQ}_2$, offering a comprehensive set of operators for simulating real-world performance issues.

C. Benchmarking Performance Mutants

This section describes how the benchmarking performance mutants will be derived.

Selecting JAVA projects with microbenchmarks. Mining all GitHub is unfeasible, given the number of projects it hosts. Therefore, we will rely on the SEART GitHub Search Engine[2] to filter projects with at least 100 stars from the set of 106,018 JAVA projects in GitHub. We

[2]Available at https://seart-ghs.si.usi.ch/

will also retain only projects where developers defined microbenchmarks for performance assessment. While other microbenchmarking tools are available (e.g., Caliper, Japex, or JUnitPerf), they are either less popular than the JAVA MICROBENCHMARKING HARNESS (JMH)[3], discontinued, or not executable in an automated way [29], [30]. Therefore, we will query the JAVA projects looking for those featuring (i) Maven as the dependency manager (i.e., contain a file named pom.xml) and (ii) a dependency on JMH, i.e., `<groupId>org.openjdk.jmh</groupId>`. As we do not aim to benchmark the performance evolution of the projects through their development, we will mine the last version of each project. In case of build failures, we will attempt to resolve them or select an older version, otherwise we will discard the project. We will select one version because mining the entire history introduces overhead without adding value to synthetic data generation for method-level analysis [31]. As a result, we will have a folder with the selected *projects with microbenchmarks*.

Selecting code covered by microbenchmarks. We will use a lightweight instrumentation agent to identify the code executed during benchmarks to gather coverage data. Methods not covered by benchmarks will be excluded, ensuring our evaluation focuses on actively tested code.

Generating performance mutants. Using GPT-4O, we will generate mutants by applying controlled transformations to degrade performance. According to Wang et al. [32], mutations generated by GPT-4O present the highest precision in detecting real functional bugs in comparison to other closed-source LLMs from GPT family and popular open source LLMs as DEEPSEEK-CODER-V2-236B and CODELLAMA-INSTRUCT-13B. Initially, we will follow the guidelines for *prompt design* when generating mutations [32], [33]. Our prompt will include: (i) *Instructions*, directing the LLM to generate mutants for the target code element; (ii) *Context*, clarifying that "mutant" refers to performance mutation and providing information such as the JAVA method (code snippets) and few-shot examples of real-world performance issues; (iii) *Input Data*, specifying the target code element and the number of mutants to generate; and (iv) *Output Indicator*, defining the JSON format for mutation outputs. Mutants will be validated to ensure: (i) They compile successfully; (ii) They do not introduce functional bugs (validated through existing test suites).

Introducing performance mutants in the code. Introducing performance mutants involves applying controlled code transformations that intentionally degrade performance, simulating real-world performance issues. Some performance mutants LLMs generate might (i) not compile or (ii) introduce functional bugs. Therefore, we will **execute the available test suites** after the injection of every mutant.

Executing microbenchmarks on the mutation. We will execute benchmarks on both the original and mutated versions of the code. Performance slowdowns caused by the mutants will be recorded as *experimental data* and compared to the *baseline data* (original code) for analysis in $\mathbf{RQ}_3$.

[3]Available at https://openjdk.java.net/projects/code-tools/jmh/.

D. Data Analysis

In $\mathbf{RQ}_0$, we aim to understand if developers use benchmarking to confirm performance issues. To this aim, we will present the percentage of issues with microbenchmarks that cover the code changed by the bug-fixing commit. The percentage will be calculated based on the performance issues covered by microbenchmarks. Complementary, we will investigate the effectiveness of the fixes in solving the performance issue. We define the following null hypothesis:

$\mathbf{H}_{0.0}$. There is *no significant difference* in performance before and after the application of the fixes.

As for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, the validation procedure is the same. GPT-4o will provide preliminary classifications and explanations for root-causes, coding patterns and mutation operators. Then, two experts will review and perform an open-coding of the explanation given by the GPT-4o. Therefore, we will calculate the Kappa statistics [34] to assess the experts' reliability. A Kappa coefficient of 0.8 or higher indicates a strong level of agreement between the two experts, demonstrating the reliability of the classification process. In addition, we will have discussions with them to solve disagreements.

As for $\mathbf{RQ}_3$, we first formulate the working hypotheses that we will later statistically assess. In $\mathbf{RQ}_{3.1}$, we investigate the impact of different performance mutation operators within a specific context, i.e., the specific code structure where a mutation operator is introduced. Differently, in $\mathbf{RQ}_{3.2}$, we analyze the impact of a specific mutation operator across a set of contexts. Therefore, the null hypothesis for each RQ is the following:

$\mathbf{H}_{0.1}$. There is *no significant difference* in the impact of different mutation operators in a specific context.

$\mathbf{H}_{0.2}$. There is *no significant difference* in the impact of a specific mutation operator across different contexts.

If one of the null hypotheses is statistically rejected, we will accept the corresponding alternative hypothesis, namely:

$\mathbf{H}_{a.0}$. There is *a significant difference* in performance before and after the application of the fixes.

$\mathbf{H}_{a.1}$. There is *a significant difference* in the impact of different mutation operators in a specific context.

$\mathbf{H}_{a.2}$. There is *a significant difference* in the impact of a specific mutation operator across different contexts.

We proposed to focus on execution time because microbenchmarking is usually targeted just at this metric (or related metrics like throughput). CPU usage and memory consumption could also be measured in microbenchmarking by employing profilers to attach to the JVM during the execution. However, this usually results in a noticeable overhead in execution time. Therefore, both measuring execution time and profiling microbenchmarks may alter the results. An alternative for RQ0 could be to focus on different performance metrics based on the reported issue (i.e., if an issue reports an increase in memory usage, we only profile the relevant

microbenchmarks to measure memory usage, not execution time). As we are not executing the performance mutation testing in the RQ0, the Mutation Score metric does not apply.

Statistical modeling for RQ_0 and RQ_3. We will produce paired distributions to answer both research questions. For RQ_0 we will focus on different performance metrics based on the reported issue (i.e., if an issue is reporting an increase in memory usage, we only profile the relevant microbenchmarks to measure memory usage, and not execution time). As we are not executing the performance mutation testing in the RQ_0, the *Mutation Score* metric does not apply. Therefore, we will have different paired distributions for RQ_0 referring to the execution time (in seconds) and memory usage (bytes) calculated in the bug-fixing commit and its predecessor. For RQ_3 the paired distributions refer to the execution time (in seconds), memory usage (bytes), and *Mutation Score* calculated in the baseline project and its mutated version. In the context of performance testing, the *Mutation Score* quantifies the proportion of *perf-mutants* that exhibit statistically significant slowdowns compared to the baseline, i.e., a *perf-mutant* is killed if its execution time is statistically larger than the baseline. In both RQs, we will use the non-parametric method based on the bootstrap approach [35] proposed by Kalibera and Jones [36] to rigorously assess the statistical significance of differences between the paired distributions and the magnitude of the observed effects.

To determine whether the difference between the two distributions is statistically significant, we will construct a confidence interval for the ratio of means. The hierarchical bootstrap method generates multiple resampled distributions from the original data, maintaining the hierarchical structure of the experiment (e.g., repeated runs, multiple forks, execution variability). We compute the ratio of means for the paired distributions for each bootstrap iteration. If the 95% confidence interval for the ratio of means includes the value 1, we conclude that the difference is not statistically significant. Conversely, if the confidence interval excludes 1, this provides evidence that the difference is statistically significant with 95% confidence. The magnitude of the effect will be computed as the point estimate of the ratio of means, which is calculated as the ratio of the arithmetic means of the two distributions. Along with the confidence interval, this provides a clear measure of the size of the difference and its associated uncertainty. For example, we might report that the metric in one distribution is "$5.5\% \pm 2.5\%$ faster" than in the other, with 95% confidence.

E. Publication of the Generated Dataset

The golden dataset of performance issues, corresponding code pairs (before and after fixing the performance issue), and generated mutants will be publicly available online [37]. We also plan to release the data collection and analysis scripts that we will use to perform this study.

V. Threats to validity

This section discusses the potential threats that may affect the validity of our empirical study plan.

Construct validity. This validity has three main threats that we will attempt to mitigate. The first threat concerns the criteria we will use to select software projects. Despite the efforts to standardize the building process, we might still fall into build failures. We will attempt to diagnose the reasons for the failure and try to fix them manually. We will discard the project if we cannot fix the build failure. Another threat concerns the performance metric selection. We will benchmark the performance in terms of execution time before and after applying the mutations in the code. However, other performance metrics (e.g., CPU usage and memory consumption) could be useful in evaluating performance.

Internal validity. Monitoring performance is challenging due to measurement noise [19]. To mitigate this issue in RQ_0 and RQ_3, we will employ a state-of-the-art methodology [36] to compare the paired distributions of execution times obtained for the bug-fixing commit and its predecessor. While we will manually validate the coding patterns and mutation operators for RQ_1 and RQ_2, their relevance and accuracy depend on experts' subjective judgment. We will measure their agreement level to assess the reliability of their validation process.

External validity. The main threat concerning the generalizability of the results is the selection of subject projects. We will select open-source JAVA projects, which are only a fraction of the complete picture of open-source JAVA projects. Therefore, replications of this study on a larger number of projects and in different contexts would corroborate our findings. Therefore, we will provide all materials and scripts used in this study to stimulate further research [37].

Conclusion validity. The main threat concerns the potential overlap between the evaluation subjects and the LLM prompt examples. Specifically, the LLM responses might not represent its ability to generate novel performance issues if the code used as input to generate the mutants were previously served as an example for the LLM. We will mitigate this risk by including several projects besides the Apache Foundation projects composing the *golden standard dataset* of performance issues. Additionally, we will ensure that the examples used in the prompt to guide the LLM do not overlap with the code to evaluate. Another threat concerns the factors that can significantly influence the model's responses. While the context length might lead the model to struggle to maintain coherence, the generation of multiple mutations may lead the model to overfit on particular patterns. We will use well-defined prompt structures to mitigate these threats, limit the context to relevant information, and filter out invalid outputs.

VI. Conclusions

The first goal of our study is to understand whether developers rely on performance benchmarks to identify performance slowdowns and verify whether the fixings solve the issue. After this step, we will identify and validate performance mutation

operators to support developers (i) evaluating the effectiveness of their performance tests, (ii) understanding whether a code change should be benchmarked, and (iii) developing static analysis tools to detect performance issues by detecting the patterns. We will conduct this study on open source JAVA projects containing JMH microbenchmarks. We will start by collecting a golden standard dataset for proposing the mutation operators, and expand our analysis to validate them into a large dataset of performance issues. Then, we will employ statistical approaches to address the goals of our investigation and, finally, provide actionable implications for researchers and practitioners. As an outcome of our exploratory study, we expect to provide the following key contribution:

- a taxonomy of causes and coding practices leading to performance issues and a catalog of new performance mutation operators to identify the issues;
- evidence of the usefulness of performance tests and the impact of performance mutation operators to improve performance testing in different software contexts;
- a novel and curated dataset of performance issues and their corresponding pairs of code pre- and post-fix;
- an online appendix that will provide all material and scripts employed to address the goals of the study.

REFERENCES

[1] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 369–378.

[2] J. Chen, "Performance regression detection in devops," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 206–209.

[3] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Software Testing, Verification and Reliability*, vol. 31, no. 5, p. e1728, 2021.

[4] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 237–246.

[5] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés, "Performance metamorphic testing: Motivation and challenges," in *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track*. IEEE, 2017, pp. 7–10.

[6] A. B. Sánchez, P. Delgado-Perez, S. Segura, and I. Medina-Bulo, "Performance mutation testing: Hypothesis and open questions," *Information and Software Technology*, vol. 103, pp. 159–161, 2018.

[7] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *2017 ieee international conference on software maintenance and evolution*. IEEE, 2017, pp. 341–352.

[8] Y. Chen, S. Winter, and N. Suri, "Inferring performance bug patterns from developer commits," in *2019 IEEE 30th international symposium on software reliability engineering (ISSRE)*. IEEE, 2019, pp. 70–81.

[9] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa, "How software refactoring impacts execution time," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–23, 2021.

[10] Y. Zhao, L. Xiao, A. B. Bondi, B. Chen, and Y. Liu, "A large-scale empirical study of real-life performance issues in open source projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 924–946, 2023.

[11] L. Weng, Y. Hu, P. Huang, J. Nieh, and J. Yang, "Effective performance issue diagnosis with value-assisted cost profiling," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 1–17.

[12] M. Imran, V. Cortellessa, D. Di Ruscio, R. Rubei, and L. Traini, "An empirical study on code coverage of performance testing," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 48–57.

[13] P. Temple, M. Acher, and J. Jézéquel, "Empirical assessment of multi-morphic testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1511–1527, 2021.

[14] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Search-based mutation testing to improve performance tests," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2018, pp. 316–317.

[15] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[16] R. Just, "The major mutation framework: efficient and scalable mutation analysis for java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 433–436. [Online]. Available: https://doi.org/10.1145/2610384.2628053

[17] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 537–548.

[18] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Information and Software Technology*, vol. 81, pp. 97–111, 2017.

[19] M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang, and W. Shang, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2022.

[20] Y. Chen, O. Schwahn, R. Natella, M. Bradbury, and N. Suri, "Slowcoach: Mutating code to simulate performance bugs," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 274–285.

[21] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Tandem: A taxonomy and a dataset of real-world performance bugs," *IEEE Access*, vol. 8, pp. 107 214–107 228, 2020.

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

[23] Y. Zhao, L. Xiao, and S. Wong, "A platform-agnostic framework for automatically identifying performance issue reports with heuristic linguistic patterns," *IEEE Transactions on Software Engineering*, vol. 50, no. 7, pp. 1704–1725, 2024.

[24] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1435–1446.

[25] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

[26] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 119–130.

[27] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 931–937.

[28] G. Colavito, F. Lanubile, N. Novielli, and L. Quaranta, "Leveraging gpt-like llms to automate issue labeling," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 469–480.

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

[30] P. Stefan, V. Horky, L. Bulej, and P. Tuma, "Unit testing performance in java projects: Are we there yet?" in *Proceedings of the 8th ACM/SPEC*

on International Conference on Performance Engineering, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 401–412. [Online]. Available: https://doi.org/10.1145/3030207.3030226

[31] F. Tip, J. Bell, and M. Schäfer, "Llmorpheus: Mutation testing using large language models," *arXiv preprint arXiv:2404.09952*, 2024.

[32] B. Wang, M. Chen, Y. Lin, M. Papadakis, and J. M. Zhang, "An exploratory study on using large language models for mutation testing," *arXiv preprint arXiv:2406.09843*, 2024.

[33] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. Wang, "Llmparser: An exploratory study on using large language models for log parsing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[34] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[35] A. C. Davison and D. V. Hinkley, *Bootstrap methods and their application*. Cambridge ; New York, NY, USA: Cambridge University Press, 1997.

[36] T. Kalibera and R. Jones, "Quantifying Performance Changes with Effect Size Confidence Intervals," 2020. [Online]. Available: http://arxiv.org/abs/2007.10899

[37] "Data collection and analysis," 2024, Accessed on 12.07.2024. [Online]. Available: https://figshare.com/s/4cd757e856febe54c0cf