Leveraging N-Version Testing to Define Approximate Oracles for Performance Testing

Críssia Marcelino
Federal University of Pernambuco
Recife, Brazil
csm4@cin.ufpe.br

Breno Miranda
Federal University of Pernambuco
Recife, Brazil
bafm@cin.ufpe.br

ABSTRACT

Performance testing plays a critical role in maintaining software quality by ensuring systems meet their expected efficiency and responsiveness. However, defining precise test oracles for performance testing remains a significant challenge. As a result, many software projects lack reliable performance test oracles, hindering the development of comprehensive test suites. Approximate test oracles have emerged as a promising alternative, offering practical means of validation in the absence of exact specifications. In this work, we explore the use of n-version testing, a technique traditionally used for fault detection through the comparison of multiple system versions, as a foundation for constructing approximate performance test oracles. Our approach leverages the performance history of recent versions of the system under test (SUT) to define an acceptable performance range. Testers configure key parameters such as the number of prior versions to consider, the strategy for computing reference performance, and the tolerance margin. When the current version's performance falls outside the derived tolerance band, an alert is raised to trigger further investigation. In our preliminary investigation using a real-world proprietary software system (an image gallery application), we used historical performance data to demonstrate that our proposed approach would have been capable of detecting a known performance bug, previously confirmed by the development team.

KEYWORDS

Performance, Test Oracle, N-version testing

1 Introduction

The continuous improvement of software is necessary to ensure it remains useful and attractive to users. Keeping software active and up-to-date in the market is not easy, requiring increasingly frequent updates—without compromising performance, of course. However, recent studies warn that constant software updates carry an almost certain risk: performance regression. Some studies have even found that most failures encountered today are related to performance rather than functional bugs [5]. Mapping these failures is essential, but fixing them is challenging. In general, perceptions of good or bad performance are subjective, requiring that the definition of requirements and expected inputs/outputs be as precise as possible.

A practice that should be indispensable in the software development process is the periodic execution of tests to ensure system stability with each new implementation. Testing is crucial for early error detection, reducing the occurrence of unplanned costs. However, in many projects, running test cycles with every new implementation is not feasible.

An alternative to streamline test execution is automation. The core artifact of this activity is the test case, which typically defines three stages: input data, execution actions, and expected output data. Yet, in many cases, output data poses challenges in confirming the success or failure of the System Under Test (SUT). According to Barr et al. [2], *"automating the generation of expected outputs is a problem that, one might argue, has been less solved than automating input generation"*. This statement highlights a weakness in automated testing outputs. In performance testing, expected outputs may incorrectly classify results since attributes like CPU, memory processing, and RAM can interfere with outcomes. One practice that aids in correctly defining expected outputs is the use of Test Oracles. A test oracle is a document or software that enables testers to evaluate whether a given test case execution has passed or failed, that is, whether its produced output matches the expected output (defined in the oracle) [4]. Thus, a key challenge in applying test oracles to automated testing is ensuring the output conforms to expectations.

If the creation/automation of Test Oracles is already a challenge for functional testing, it stands to reason that for non-functional aspects such as Performance, output definitions are even more complex. In performance testing, system efficiency metrics can reveal degradation errors [11], such as when a server fails to respond under high concurrent request loads. Thus, it follows that one of the key challenges in modern software projects with continuous integration is ensuring that performance quality is preserved, as well as establishing mechanisms to detect output deviations caused by Performance regression. A common practice in such projects is relying on testers to certify the software, using test results to build evidence that legitimizes the system's performance.

In many project contexts, testers face the daunting task of manually verifying the system's expected behavior for all test cases [1], making the validation process significantly more time-consuming.

Given this scenario, it is essential to adopt an approach that enables greater objectivity in test execution while reducing personnel effort and improving result accuracy. Thus, we propose that creating an approximate test oracle focused on Performance testing (a non-functional property), based on the N-version testing method, is a promising approach. The combination of N-version testing + Test Oracle falls under the category of Derived Test Oracles, which utilize artifacts such as documentation, system executions, or properties of the system under test (or other versions of it) [2].

For this study, we selected regression testing as the most effective approach for creating efficient test oracles to detect Performance-related errors. Regression testing aims to identify whether new software implementations compromise existing functionality [16]. It relies on the implicit assumption that the previous version can serve as an oracle for existing functionality. Therefore, this oracle helps establish reliable guidelines for projects lacking clearly defined performance metrics, enhancing precision and consistency in software quality assessment.

The remainder of this paper is organized as follows. Section 2 provides the necessary background on performance, performance regression, test oracles, regression testing, and N-version testing. Section 3 outlines the methodology employed in our study. Section 4 describes the experimental context, the artifacts used, the steps followed for executing the tests, and our prelimnary results. Finally, Section 5 summarizes our findings and concludes the paper.

2 Background

2.1 Performance

Performance is a non-functional requirement that pertains to software quality attributes. It measures the ability of a system or application to efficiently respond to user requests, considering criteria such as response time, computational resource usage, and scalability. According to Liao [10], performance evaluates the effectiveness of a software system through various metrics, including response time, throughput, and CPU utilization. Thus, regardless of the intended purpose of a given software, it must deliver rapid responses without excessive hardware resource consumption, ensuring user satisfaction and system scalability.

Studies indicate that non-functional aspects are more likely to exhibit field failures in large-scale software than functional aspects [6]. Among these, performance demands particular attention due to its potential impact, often determining an application's success or failure. Consequently, performance analysis should be incorporated from the early stages of software development, as late-stage optimization changes tend to be costlier and less effective [3].

As a common and critical type of performance issue, performance regressions occur when a system's new version remains functionally correct but delivers a degraded user experience (e.g., slower response times) and/or consumes additional resources (e.g., memory leaks) compared to previous versions. Such regressions can reduce user satisfaction, increase operational costs, and lead to field failures [10].

2.2 Performance Regression

Performance regression occurs when a new version of software performs worse than a previous version in terms of response time, which is the interval between a request and its respective response, throughput, which measures the number of operations completed per unit of time, resource consumption or scalability. This can compromise the user experience and increase the system's operating costs. According to Smith [12], *"a performance regression occurs when modifications to the software result in a worsening of efficiency, even without apparent functional change"*.

Basically, any change to a version, a function or a bug fix can inadvertently introduce new problems. One of the main practices for identifying potential flaws is to apply regression testing to the software. The aim of regression testing is to ensure that modifications do not affect software features that already work efficiently. Although a new version may appear to be functionally sound, it may suffer from degraded performance, such as increased resource utilization or other efficiency drawbacks [7].

In view of the studies analyzed, a viable solution proposed by this work is to define an approximate performance test oracle, as a result of regression tests, to guide the team in identifying potential regression errors. We identified that the use of Oracles as a control tool for performance regression is still being explored in a timid way in software testing studies. We therefore intend to show the efficiency and assertiveness of this approach in regression testing.

2.3 Test Oracle

The test oracle can be defined as a mechanism that establishes acceptable thresholds for response times and resource consumption. For instance, in a web system required to respond within 200ms for 95 percent of requests, the oracle can be programmed to automatically flag any values exceeding this threshold as performance test failures. In software performance testing, this oracle serves to validate whether performance requirements are being met by defining acceptance criteria for key metrics such as response time, throughput, and resource utilization. By comparing current results against previously established benchmarks, the oracle enables objective performance evaluation. Crucially, it can detect performance regressions by identifying when a new software version demonstrates degraded performance compared to previous versions, providing an automated means to verify that updates maintain expected performance standards. This approach transforms subjective performance assessment into a measurable, repeatable testing process.

The oracle can also support automation and continuous monitoring by automatically validating load and stress tests, or by being integrated into CI/CD pipelines to prevent the deployment of versions with performance issues. Another key strength of oracles is their ability to identify bottlenecks. By defining expected values for CPU usage, memory consumption, and latency, the oracle helps detect resource usage anomalies.

According to Barr et al. [2], test oracles can be classified into three categories: Specified test oracles, which evaluate all behavioral aspects of a system against formal specifications (i.e., they rely on documentation). Derived test oracles, which are based on artifacts from which an oracle can be derived—such as a previous version of the system. Implicit test oracles, which detect "obvious" failures, such as program crashes.

Since this research aims to identify performance regression through N-version testing, the most appropriate category for this study is the derived test oracle. A derived test oracle distinguishes correct from incorrect system behavior based on information derived from various artifacts (e.g., documentation, system executions) or properties of the system under test, including other versions of it. Within this context lies the Regression Test Suite, which serves as a practical implementation of this approach.

2.4 Regression Testing

Regression testing is a software testing approach that verifies whether source code changes - such as implementing new features, fixing defects, or refactoring activities - have introduced faults into previously working system components. This technique is based on the implicit principle that the previous software version can serve as an oracle to validate existing functionality. As emphasized by

Sommerville [13], *"regression tests ensure that software modifications do not adversely affect previously validated system behaviors"*

Regression tests are executed through test cases that detail each step to be performed and the expected behavior in a given context. These test cases consist of two fundamental elements: on one hand, the input data required to exercise the program under test; on the other, a test oracle that allows verifying the correctness of test execution, as explained by Xie [15]. When applied to performance evaluation, the results obtained from running regression test cycles serve as a crucial indicator of software stability, enabling assessment of whether performance parameters were maintained or if degradation occurred following system modifications.

2.5 N-version Testing

Therefore, a promising approach to verify whether code behavior from previous versions remains consistent with its implementation is the use of N-version testing. This method detects functional problems shared between a new system version and its predecessors, serving as a precursor to derived test oracles. In this technique, N different software versions are independently developed to perform the same function. These versions are then executed in parallel, with their outputs compared to detect inconsistencies. The primary objective of this approach is to enhance software reliability and robustness while minimizing the impact of failures introduced by implementation errors.

3 Proposed Approach

Performance metrics can be measured in different ways, even for the same metric, and the measurement method directly influences how the data is described and analyzed. In the literature [8, 9, 14], two primary approaches to measuring performance metrics are identified: Point-in-Time Measurement and Time-Series Measurement.

- **Point-in-Time Measurement:** Captures the value of a metric at a specific moment in time, typically used to assess the system's state after a specific action or event.
  Example: "Memory usage was 1.3 GB right after the file was loaded."
- **Time-Series Measurement:** Involves collecting a sequence of metric values at regular or irregular intervals over time, enabling the analysis of trends, patterns, and dynamic behavior.
  Example: "Memory usage was monitored every second for 5 seconds during the test, resulting in a time-series plot."

Regardless of the measurement type (point-in-time or time-series), our approach, leveraging **n-version testing** to define **approximate performance oracles**, follows the same four-step process. An overview of the proposed approach is shown in Figure 1.

Step 1 – Preparation: Define Metric and Collect Historical Data The performance metric of interest is specified, and historical data is collected across *n* prior versions of the SUT. The performance expert selects how many versions to include (e.g., the last 10 or 20).

Step 2 – Preparation: Establish Reference Performance Using the collected historical data, a **Reference Performance** baseline is computed.

For point-in-time metrics, this could be a single value (e.g., average or median) or a range (e.g., minimum and maximum values).

For time-series metrics, a curve can be derived representing the average or median value at each time point across the *n* versions; minimum and maximum curves can also be computed to represent the range of behavior at each time step.

This step incorporates principles from N-version Testing, treating each prior version as an independent source of performance behavior.

Step 3 – Preparation: Define Tolerance Margin A **Tolerance Margin** is applied to the reference performance to form an approximate oracle. This margin can be defined using standard deviation, fixed thresholds, or percentages over the baseline (mean, median, min, or max). The margin specifies acceptable deviation before signaling a performance regression.

Step 4 – Execution: Run Test and Compare Results The test case is executed on the current version of the SUT using the same measurement method applied to previous versions. The resulting metric is then compared against the reference performance and its tolerance margin:

- If the observed value falls **within** the margin, the system is assumed to behave as expected.
- If the value is **outside** the margin, an alert is raised, indicating a potential performance regression requiring further investigation.

Next, we demonstrate this four-step process with two illustrative examples: one using a point-in-time metric and another using a time-series metric. Please note that the possible combinations of parameters (i.e., number of historical versions, the reference performance, and the tolerance margin) are diverse. These combinations depend on factors such as the type of system, the dynamics of the project, and the specific scenario being tested. Therefore, we emphasize the importance of involving an expert with experience in the software's performance requirements and characteristics.

3.1 Example with Point-in-Time Measurement

This example illustrates how our four-step approach is applied when dealing with point-in-time performance metrics, specifically Memory usage for our example (see Figure 2).

In **Step 1** (Define Metric and Collect Historical Data), memory usage was selected as the performance metric of interest. Data was collected at a specific point in time, immediately after a defined software operation (e.g., load a file), for 10 previous versions of the SUT (Figure 2a).

In **Step 2** (Establish Reference Performance), a reference value was computed by calculating the mean memory usage across the 10 historical versions. This value represents the expected memory consumption at that specific point in the operation (Figure 2b).

In **Step 3** (Define Tolerance Margin), a tolerance range was derived by applying a margin of two standard deviations above and below the reference mean. This range defines acceptable variability and acts as an approximate performance oracle (Figure 2c).

In **Step 4** (Run Test and Compare Results), memory usage was measured for the current version of the SUT (version 11) using

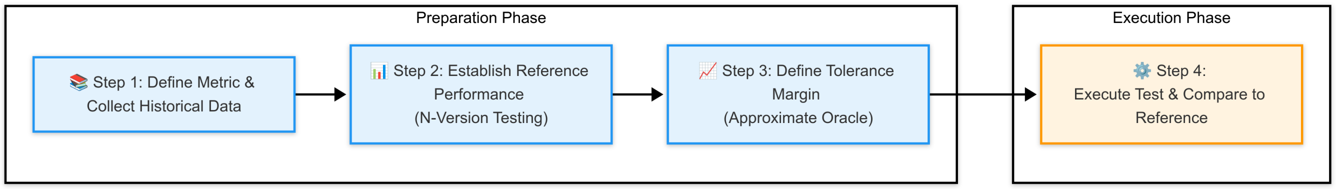
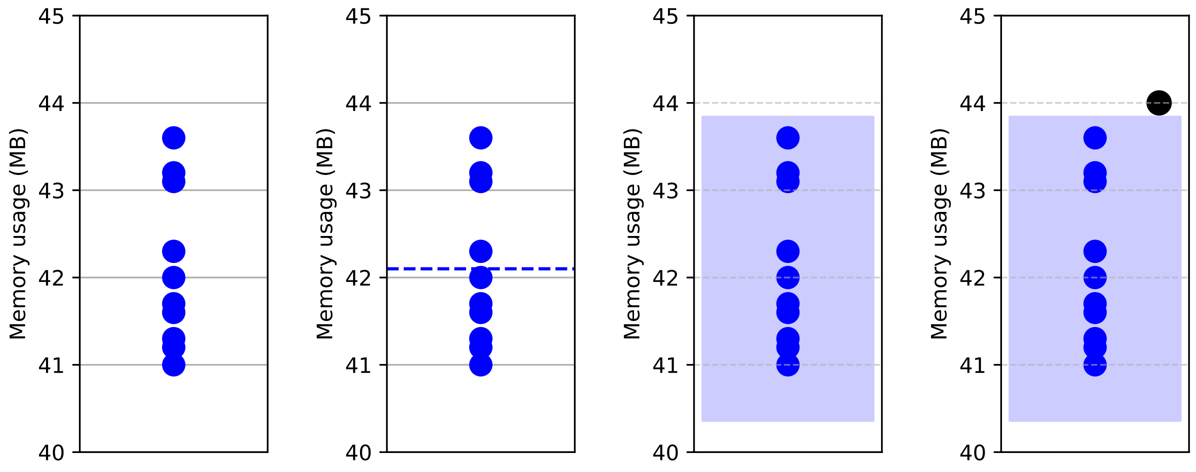

Preparation Phase
Step 1: Define Metric & Collect Historical Data
Step 2: Establish Reference Performance (N-Version Testing)
Step 3: Define Tolerance Margin (Approximate Oracle)
Execution Phase
Step 4: Execute Test & Compare to Reference

Figure 1: Four-step approach for defining approximate performance oracles based on N-Version testing



(a) Performance from prior versions **(b) Reference performance** **(c) Tolerance margin** **(d) Current performance assessment**

The dashed blue line represents the average, the shaded purple area indicates the tolerance margin and the black dot in the graph (d) indicates the performance in version 11.

Figure 2: Example of applying the four-step approach to point-in-time Memory usage data across 10 software versions

the same method. The observed value was compared to the previously defined tolerance range. Since the result exceeded the upper limit of the range, an alert was raised, indicating a potential performance regression in memory consumption that warrants further investigation (Figure 2d).

3.2 Example with Time-Series Measurement

This example illustrates how our four-step approach is applied when dealing with time-series performance metrics, specifically CPU utilization for our example (see Figure 3).

In **Step 1** (Define Metric and Collect Historical Data), CPU usage was chosen as the performance metric of interest. Measurements were taken every 0.5 seconds over a 5-second period, across 10 previous versions of the SUT (Figure 3a).

In **Step 2** (Establish Reference Performance), a reference curve was computed by calculating the mean CPU usage at each time interval across the 10 versions. This reference curve represents the expected behavior over time. Although it is also possible to define minimum and maximum bounds, in this example, we opted to use only the average curve as the reference performance (Figure 3b).

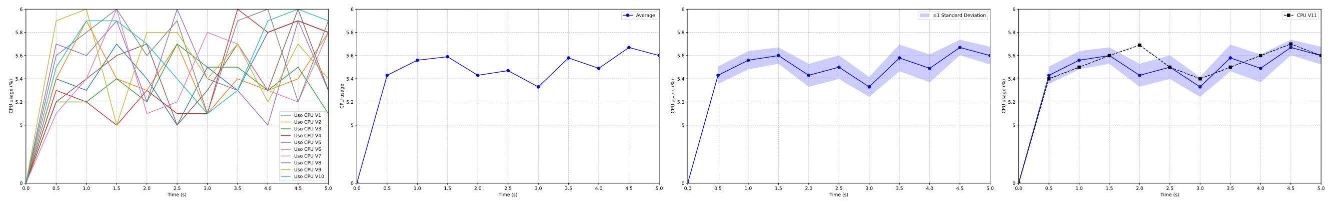
In **Step 3** (Define Tolerance Margin), the tolerance area was constructed by adding and subtracting 1 standard deviation from the reference curve at each time point. This creates a bounded area representing acceptable variability based on historical behavior (Figure 3c).

Finally, in **Step 4** (Run Test and Compare Results), the same test was executed on the current version of the SUT (version 11), collecting CPU usage at the same time intervals. The resulting curve was compared against the tolerance area. Data points falling outside the defined bounds indicate a potential performance regression and signal the need for further investigation to determine the cause of the deviation (Figure 3d).

4 Preliminary Assessment

For the experimental context of this study, we opted for discrete data collection, as the analyzed action is specific and involves only recording the execution time of a particular system operation. The obtained data was then organized into charts illustrating all stages of this analytical framework.

For the execution of this experiment, an industrial application was selected. While confidentiality restrictions prevent us from disclosing the application's identity, we emphasize that it is widely used for image editing in both desktop and mobile environments. This application was chosen specifically because it has documented records of performance regression bugs in its project management tool, making it a suitable model for collecting and analyzing performance results. The study scope reflects real-world conditions



(a) Performance from prior versions **(b) Reference performance** **(c) Tolerance margin** **(d) Current performance assessment**

Figure 3: Example of applying the four-step approach to time-series CPU usage data across 10 software versions

faced by users operating in continuous integration environments, enabling a more practical evaluation. The error scenario selected for reproduction corresponds to an actual, previously reported and fixed issue in the application, ensuring data authenticity and analytical relevance.

The error description indicates that when selecting a thumbnail image to enlarge its view, the time to fully display the image reached 15.6 seconds—in stark contrast to previous versions where this process never exceeded 2 to 3 seconds. This reference value was subjectively estimated by the testing team without concrete validation or pre-established metrics. Notably, the development team promptly acknowledged the issue. However, the debate centered on defining an acceptable response time, as no formal performance requirement existed for this functionality.

The lack of clearly defined performance requirements or key performance indicators (KPIs) in this project motivated the interest in establishing an approximate performance test oracle, using the historical performance of prior software versions as a reference. In this context, we identified that applying the n-version testing method could be an effective approach for measuring consistent output data, enabling a comparative and systematic assessment of the application's behavior across its different versions.

4.1 Preparation

To apply our approach in a preliminary evaluation of a real-world application, we selected an issue that identified a version with a previously confirmed performance regression problem. After reserving this version as a reference, we retrieved the ten preceding versions of the application and applied our approach in successive execution steps.

The performance issue manifested when enlarging a 1920x1080 pixel image (227 KB in size). The loading time for this image, which in preliminary versions varied between 2 and 3 seconds, increased to 15 seconds in the regressed version. Notably, resolving this bug required extensive discussion among testers, developers, and project leaders, as no established performance metrics existed to define acceptable response times for this (or any) application operation. Given this absence of criteria, the adopted solution used previous versions' execution times as the performance benchmark.

Our approach was implemented in the replication scenario as follows: In **Step 1** (Define Metric and Collect Historical Data), ten previous system versions were selected to collect the 'response time' metric after operation execution (Figure 4a). In **Step 2** (Establish Reference Performance), the collected data were used to derive the performance baseline (Figure 4b). Subsequently, in **Step 3** (Define Tolerance Margin), a tolerance margin was defined based on two standard deviations from the reference line, as illustrated in Figure 4c. Finally, in **Step 4** (Run Test and Compare Results), the test was executed on the target version, which exhibited a known latency issue, and the results were plotted on the same graph (Figure 4d), enabling clear analysis of the anomalous behavior displayed by this version.

The combination of parameters (10 historical builds, the average as the performance reference, and 2 standard deviations for the tolerance margin) was defined by a test specialist, a member of the project being evaluated. The specialist considers this combination to be a balanced and realistic metric for the scenario being tested.

Based on the results obtained thus far, we begin our analysis of the oracle's relevance and effectiveness in identifying performance failures in projects lacking clear requirements definition for such parameters.

4.2 Execution

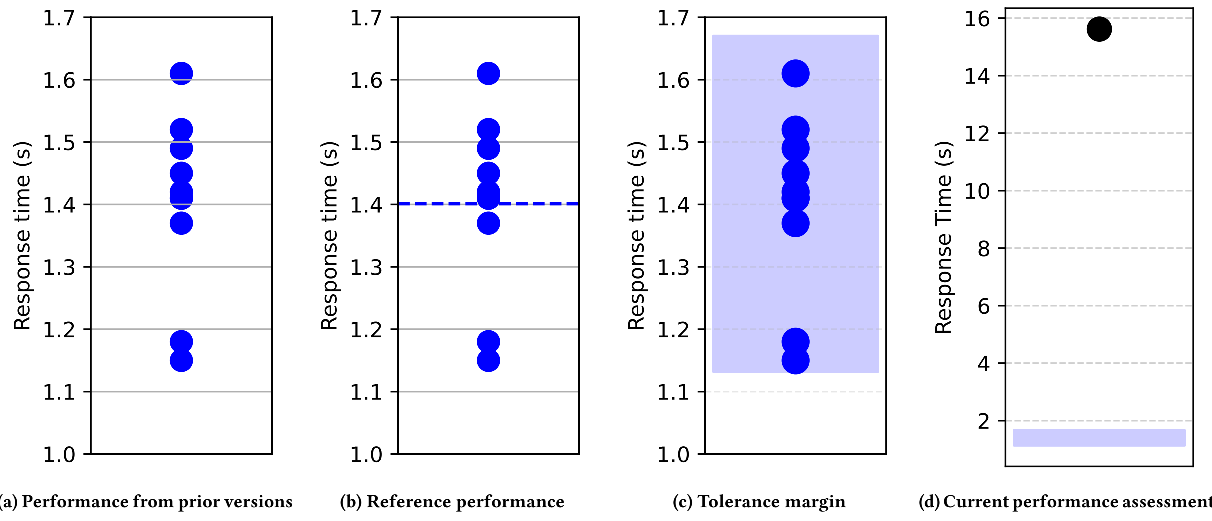
We automated the test using Python within Visual Studio Code (VSCode), conducting all experiments sequentially on a single machine to minimize external interference that could compromise results. The test procedure involved: (1) launching the application via Windows navigation bar search; (2) locating a specific image using the application's search field; and (3) selecting the thumbnail from search results. An automated script then recorded the response time until full image display.

The sequential visualization in Figure 4 demonstrates a empirical approach to performance regression detection. The four-panel structure systematically progresses from raw data collection (a) to statistical benchmarking (b-c) and anomaly detection (d), effectively operationalizing the concept of metric-driven quality control.

The figure 4(a) provides a visual representation of the historical distribution of response times, enabling the identification of variability and performance trends. This step is crucial for establishing a baseline—a reference performance pattern derived from empirical data.

The figure 4(b) presents the calculation of the overall mean response time. This value serves as a global reference metric, representing the typical behavior of the system throughout the evaluated versions. In practice, this mean may be integrated as an acceptance criterion in regression testing, reinforcing traceability and performance predictabilit.

The figure 4(c) introduces a tolerance band based on the standard deviation of the general mean ($\pm 2\sigma$). By visually emphasizing this margin, the figure illustrates an acceptable variation range



The dashed blue line represents the average, the shaded purple area indicates the tolerance margin and the black dot in the graph (d) indicates the performance in version 11.

Figure 4: Applying the four-step approach to point-in-time Response Time in a real application with known performance bug

in response times, reflecting the natural capability of the development process. This approach prevents false alarms due to minor fluctuations and focuses attention on significant deviations, thus promoting a more robust performance verification process.

The figure 4(d) illustrates the introduction of a new performance measurement (V11), directly compared against the previously established tolerance range. The V11 value lies outside the standard deviation band, indicating a serious performance regression.

The figure clearly demonstrates how the definition of statistical thresholds can support technical decision-making in software evolution. Using performance benchmarks and tolerance margins as quantitative criteria not only facilitates automated performance testing, but also strengthens governance over software quality. Versions with values outside the defined range become candidates for inspection, supporting a continuous improvement culture and preventing the silent introduction of performance-related defects.

This preliminary evaluation seems to support the use of tolerance margins for comparative assessments. A well-defined tolerance margin serves as an effective gauge for: (1) prioritizing issues, (2) evaluating inconsistency severity, (3) enabling preemptive resolutions, and (4) reducing unplanned corrective costs.

5 Conclusion and Future Work

The obtained results prompt a discussion about the importance and effectiveness of test oracles in detecting performance issues. These initial findings highlight the value of establishing test oracles for requirements lacking defined metrics, helping teams monitor performance variations. It is crucial to clarify that this experiment does not categorically classify results outside the established tolerance range as performance errors; rather, it signals that such inconsistencies warrant detailed analysis before potential defects are identified too late.

To further validate this approach, we plan to apply n-version testing to open-source applications with documented performance issues in their repositories. Additionally, we intend to expand the analysis to include other performance metrics such as energy consumption, network usage, data transfer rates, and temperature measurements. The acceptable tolerance margin could also be modeled using alternative statistical approaches, including the addition of confidence intervals to reference calculations and the incorporation of non-parametric methods.

Should the results confirm our hypothesis, this approach would make the test oracle highly recommendable for performance-sensitive applications, continuous integration pipelines, and systems with frequent iterative updates.

ARTIFACT AVAILABILITY

The artifacts used in our work for preliminary evaluation are derived from proprietary software and are subject to industrial secrets, so they cannot be shared.

ACKNOWLEDGMENTS

This work was supported by Samsung Eletronica da Amazônia Ltd., through the Collaborative Project between SiDi and the Centre of Informatics, Federal University of Pernambuco (CIn/UFPE), through the auspices of the Brazilian Federal Law of Informatics, under Grant 8248/91. This work was partially supported by a grant from the National Council for Scientific and Technological Development (Grant CNPq-Universal 408651/2023-7).

REFERENCES

[1] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 2013. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 352–361. https://doi.org/10.1109/ICST.2013.11
[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785
[3] Clements P. Kazman R Bass, L. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley.

[4] Ilene Burnstein. 2006. *Practical software testing: a process-oriented approach* (1st ed.). Springer Science Business Media.

[5] Jinfu Chen and Weiyi Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 341–352. https://doi.org/10.1109/ICSME.2017.13

[6] Jinfu Chen and Weiyi Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 341–352. https://doi.org/10.1109/ICSME.2017.13

[7] Luciana Brasil Rebelo dos Santos, Érica Ferreira de Souza, André Takeshi Endo, Catia Trubiani, Riccardo Pinciroli, and Nandamudi Lankalapalli Vijaykumar. 2025. Performance regression testing initiatives: a systematic mapping. *Information and Software Technology* 179 (2025), 107641. https://doi.org/10.1016/j.infsof.2024.107641

[8] Henrik Ingo and David Daly. 2020. Automated system performance testing at MongoDB. In *Proceedings of the workshop on Testing Database Systems*. 1–6.

[9] Mark Leznik, Md Shahriar Iqbal, Igor Trubin, Arne Lochner, Pooyan Jamshidi, and André Bauer. 2022. Change point detection for MongoDB time series performance regression. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. 45–48.

[10] Lizhi Liao. 2023. Addressing Performance Regressions in DevOps: Can We Escape from System Performance Testing?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 203–207. https://doi.org/10.1109/ICSE-Companion58688.2023.00056

[11] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. 2013. Automatic detection of performance deviations in the load testing of Large Scale Systems. In *2013 35th International Conference on Software Engineering (ICSE)*. 1012–1021. https://doi.org/10.1109/ICSE.2013.6606651

[12] Williams L. G Smith, C. U. 2002. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* (1st ed.). Addison-Wesley.

[13] I. Sommerville. 2011. *Software Engineering* (9th ed.). Addison-Wesley.

[14] Luca Traini, Federico Di Menna, and Vittorio Cortellessa. 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*. 443–454.

[15] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *ECOOP 2006 – Object-Oriented Programming*, Dave Thomas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 380–403.

[16] Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 65–120. https://doi.org/10.1002/stvr.1466