An Empirical Investigation on the use of Large Language Models for Performance Bug Detection

Muhammad Imran, Vittorio Cortellessa, Davide Di Ruscio, Riccardo Rubei†, Luca Traini
University of L'Aquila, L'Aquila, Italy
Emails: muhammad.imran@graduate.univaq.it, vittorio.cortellessa@univaq.it, davide.diruscio@univaq.it, luca.traini@univaq.it
†*Mälardalen University*, Västerås, Sweden
Email: riccardo.rubei@mdu.se

***Abstract*—Performance bugs are non-functional defects that significantly impact software performance. Identifying such bugs can be challenging, as they are typically harder to detect than other types of software defects and often require specialized expertise that may not be readily available within software organizations.**

Previous approaches have attempted to automate performance bug detection using static analysis or traditional machine learning models trained on static code metrics. However, despite the growing potential and widespread adoption of Large Language Models (LLMs) for automating various software engineering tasks, no studies have directly investigated their capabilities in detecting performance bugs.

In this paper, we aim to fill this gap by exploring the potential of LLMs–such as CodeLlama, Qwen, and Artigenz–for detecting performance bugs directly from source code. We focus on zero-shot and few-shot prompting, as well as supervised fine-tuning, to evaluate these models using Java code from open-source projects with labeled performance bugs. Our results highlight the limitations of current LLMs in this domain: they achieved low F1 scores, with few-shot prompting providing only marginal improvements over zero-shot configurations, and fine-tuning yielding slight gains.

***Index Terms*—Performance Bugs, Large Language Models, Machine Learning, Java, Bug Detection**

I. Introduction

Software performance is a critical non-functional attribute that directly impacts user satisfaction and system efficiency. However, software systems can sometimes experience severe efficiency degradation due to performance bugs, which are programming errors that can affect software performance, such as memory leaks, redundant computations, and inefficient resource utilization [1].

Figures 1 and 2 show two examples of these types of bugs. Figure 1 shows a memory leak [1], which occurs when a program fails to release memory that is no longer needed, leading to gradual resource exhaustion that reduces system performance. In this example, the `events()` method results in unreleased file handles and memory consumption, as the `DataInputStream` object is not properly closed when an exception occurs or if the stream is not explicitly closed.

This is a pre-copyedited, author-produced version of an article accepted for publication in the *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2026)*.
Citation information: DOI 10.1109/SANER67736.2026.00044

```
48 public Stream<AccessEvent> events() {
49   var input = new DataInputStream(readFile());
50   var stream = StreamSupport.stream(Spliterators.spliteratorUnknownSize(
51     new TraceIterator(input), ORDERED | NONNULL), /* parallel */ false);
52   return stream.onClose(() -> Closeables.closeQuietly(input));
53 }
```

Fig. 1: An example of memory leak bug in file BinaryTraceReader.java [7] of project `caffeine`.

```
115 private Response indexAllOrSubset(Long numPartitionsSelected) {
    ....
125 List<Long> availablePartitionIds = new ArrayList<>();
126 for (long i = 0; i < numPartitions; i++) {
127     availablePartitionIds.add(i);
128 }
    ....
150 JsonArrayBuilder availablePartitionIdsBuilder = Json.createArrayBuilder();
151 for (long i : availablePartitionIds) {
152     availablePartitionIdsBuilder.add(i);
153 }
```

Fig. 2: An example of redundant traversal bug in file Index.java [8] of project `dataverse`.

Figure 2 shows an example of a redundant traversal bug [2], which arises when loops are implemented inefficiently, causing redundant iterations and excessive computational overhead. Here, the `indexAllOrSubset()` method performs unnecessary iterations on collections, leading to additional processing overhead.

Unlike functional bugs, which exhibit fail-stop symptoms, performance bugs gradually degrade software performance, making them more difficult to detect. For example, the bug shown in Figure 1 degrades memory usage as the amount of open files grows with each invocation. Similarly, the bug shown in Figure 2 causes immediate but less noticeable slowdowns, as redundant iterations increase CPU usage and response times without triggering explicit failures. These types of bugs can remain latent for long periods, posing significant challenges for detection and diagnosis. Furthermore, performance bugs are perversive in software projects, as reported by several empirical studies [3]–[6].

The traditional approach to preventing performance bugs from being released into production relies on performance testing [9]–[11]. However, this approach has several well-known limitations. First, performance tests must be prop-

erly designed to be effective, as they are subject to non-deterministic behavior [12], [13] and often require well-crafted workloads to expose performance bugs [4], [14]. Second, performance tests tend to have lower code coverage than functional tests [15], [16], which may limit their ability to detect performance bugs [17]. Finally, performance testing is often deprioritized compared to other development and quality assurance activities [18]–[20].

To overcome these limitations, researchers have developed techniques to statically detect performance bugs directly from source code, without requiring software execution. However, most of these approaches target specific types of performance bugs (e.g., redundant traversal bugs [2], [21], synchronization bugs [22], memory leaks [23]) and are unable to detect multiple bug types simultaneously. The only notable exception is the technique proposed by Zhao *et al.* [1], which leverages static code metrics to train machine learning classifiers capable of predicting whether a Java class is affected by a performance bug.

Recent advances in Large Language Models (LLMs) have enabled a wide range of possibilities for automating software engineering tasks, including defect detection, code generation, and software security analysis [24], [25]. LLMs can directly process raw source code without the need to extract code metrics, as in Zhao *et al.*'s ML-based approach [10]. This capability makes them attractive candidates for performance bug detection, particularly when labeled training data is scarce or unavailable.

Regardless of their potential, studies have shown that LLMs often struggle with software analysis tasks that require precise reasoning about the execution and performance characteristics of the program [26]–[28]. Existing studies on vulnerability detection have shown that LLMs often perform poorly compared to traditional static and dynamic analysis tools, particularly when applied in zero-shot settings or without extensive fine-tuning [29], [30]. Given the complexity of performance bugs, it remains uncertain whether LLMs can effectively identify them.

In this study, we empirically evaluate the effectiveness of LLMs in detecting performance bugs directly from source code by leveraging their text processing capabilities, without relying on any code metric extraction from the source code. We investigate their capabilities in zero-shot and few-shot settings, as well using fine-tuning, comparing their performance with ML-based methods. Our evaluation focuses on 31 open-source Java projects crawled from GitHub, allowing us to observe LLMs at work on real-world software systems. Our findings reveal that LLMs struggle to accurately detect performance bugs, achieving low F1 scores even with few-shot prompting. These results contribute to this growing field of research on LLMs in software engineering by highlighting their current limitations. In line with recent reporting guidelines [31], we explicitly note that this study uses LLMs as subjects to perform a binary classification task, identifying performance bugs directly from Java source code without program execution.

The main contributions of this paper are:

- A first empirical evaluation of LLMs for performance bug detection, involving different prompting strategies, such as zero-shot and few-shot, and supervised fine-tuning.
- A comparative analysis on the effectiveness of LLMs and traditional ML-based models in performance bug detection.
- A publicly available replication package [32], including the dataset, the scripts we used to perform the experiments and analyze the results, and the detailed results of our analysis.

II. Study Design

This study investigates the effectiveness of LLMs in detecting performance bugs in source code. Besides assessing the performance bug detection capabilities of LLMs, we also perform a comparative analysis with traditional ML-based approaches which involve model training on predefined code metrics.

For the evaluation, we use the dataset from Zhao *et al.* [1], which includes Java source code files labeled as *buggy* or *non-buggy*. Specifically, a Java file is considered *buggy* if it has been modified by a commit explicitly linked to a performance bug report or fix in the project's issue tracker, following the labeling procedure employed in their study [1]. Files without such links are considered *non-buggy*. The performance bugs span multiple categories (e.g., memory leaks, redundant traversals, infinite loops, and regressions). The Java source code files included in the dataset originate from well-established and actively maintained projects selected according to criteria ensuring their relevance for performance bug detection–such as the presence of detailed bug reports and a sufficient number of performance bug–fixing commits.

In this study, our objective is to address the following research questions (RQ).

▷ **RQ$_1$**: *How accurate are LLMs in detecting performance bugs in a zero-shot setting?* Our objective is to evaluate the ability of LLMs to detect performance bugs when applied directly to source code. We assess their performance using zero-shot prompts, relying solely on pre-training knowledge without supervised fine-tuning or exposure to labeled examples. Additionally, we examine whether comments in the source code influence performance and include a project-wise analysis to understand performance variability across projects.

▷ **RQ$_2$**: *What is the impact of using few-shot prompts on the effectiveness of LLMs in detecting performance bugs?* We aim to determine whether using few-shot prompts impacts the ability of LLMs to detect performance bugs in source code. We assess their performance by providing one labeled buggy file and one labeled non-buggy file as contextual guidance in the few-shot examples. Additionally, we examine whether comments in the source code influence performance and include a project-wise analysis to explore performance variability across projects.

▷ **RQ**$_3$: *What is the impact of using supervised fine-tuning on the effectiveness of LLMs in detecting performance bugs?* We evaluate the effect of fine-tuning on the ability of LLMs to detect performance bugs in source code. For this purpose, we fine-tune Qwen2.5-Coder-7B using QLoRA on our dataset and assess its performance against zero-shot settings.

▷ **RQ**$_4$: *How do LLMs compare to traditional machine learning models in detecting performance bugs?* This research question compares the performance of LLMs with traditional machine learning models in detecting performance bugs. We focus on the best-performing configuration of each LLM and use the Wilcoxon signed-rank test [33] to analyze differences in model performance, focusing on F1 scores.

Our study follows a structured execution process, shown in Figure 3, which consists of four main steps: *(i) data preparation*, *(ii) LLM inference*, *(iii) LLM fine-tuning*, and *(iv) machine learning prediction*. Each of these steps is described in the following subsections.

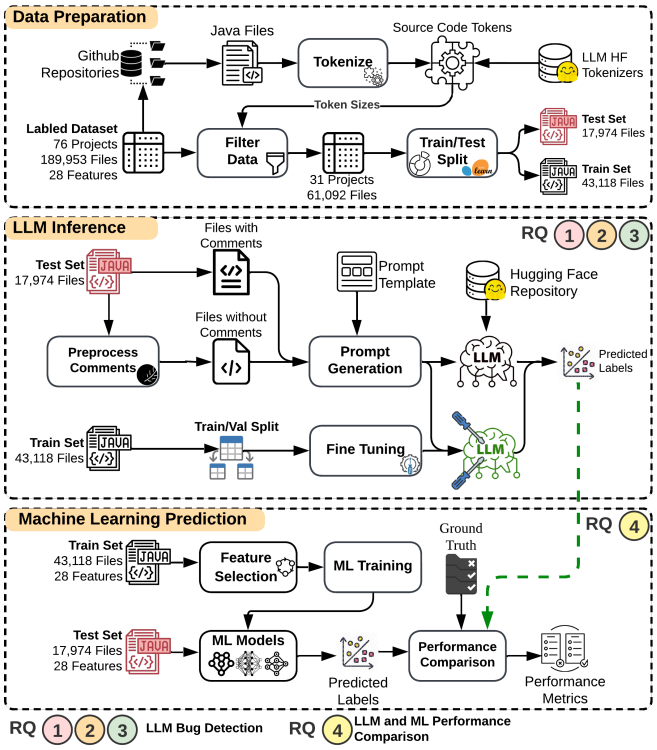

Fig. 3: Main steps of the performed study.

1) ***Data Preparation****:* Our data preparation involves processing Java source files from the selected dataset to ensure they are in a suitable format for both LLM inference and machine learning predictions. This process consists of filtering projects, tokenizing source code, and splitting training and testing sets.

The dataset used in our study was derived from [1], which initially included 76 well-established Java projects with labeled performance bugs. However, due to repository availability issues, some projects could not be retrieved using the metadata provided, thus they were excluded. Additionally, we excluded projects without at least two positively labeled buggy files. This ensured a sufficient number of positive samples for stratified splitting.

Furthermore, to ensure compatibility with LLMs and avoid input truncation during the inference phase of LLMs, we determined the file size by tokenizing the Java source code files beforehand through model-specific tokenizers accessing from the Hugging Face repository. Files exceeding the context length of the respective LLMs were dropped, ensuring that only completely processable files were retained for further steps. This resulted in a final dataset of 31 projects with 61,092 Java source files.

Finally, the dataset was divided into training and test sets, following a stratified approach using scikit-learn [34] to preserve the distribution of buggy and non-buggy files. The final training set consists of 43,118 ($\approx$ 70%) files, whereas the test set contains 17,974 ($\approx$ 30%) files, ensuring that both sets maintain a representative distribution of buggy and non-buggy files ($\approx$ 1.2% buggy, $\approx$ 98.8% non-buggy). This stratification ensures a reliable evaluation of model performance while preserving the relative proportions of buggy files across the splits.

2) ***LLM Inference****:* To evaluate the accuracy of LLMs in detecting performance bugs, we conducted two separate experiments: one using source code files including comments and another one using the same files with comments removed. Comments were removed using `srcML`, which converts source code to XML for structural analysis while preserving layout. Additionally, for LLM inference, we only utilized files from the test set since LLMs are pre-trained and do not require a separate training phase.

The selection of LLMs was guided by four key considerations: *(i)* the model size (`7B`) was considered to ensure the feasibility of running the experiments given limited computational resources; *(ii)* models were required to have `instruct` versions available or to support chat template for inference, enabling the use of chat-based interaction with system and user roles, as well as few-shot prompting; (iii) the selected models had to be pre-trained on code to facilitate reliable performance bug detection in the source code, and (iv) they were employed by previous studies for tasks related to source code assessment and bug detection [27], [29], [35]. Based on these criteria, we selected the following LLMs for our study: CodeLlama 7b instruct [36], Qwen Coder 7B instruct [37], and Artigenz Coder 6.7B [38]. For reproducibility, we report the model snapshots used in our runs: *codellama/CodeLlama-7b-Instruct-hf*, *Qwen/Qwen2.5-Coder-7B-Instruct*, and *Artigenz/Artigenz-Coder-DS-6.7B*. Experiments were executed on *1× NVIDIA A30 24 GB GPU, 504 Gi system RAM*. All three evaluated models are open source, which satisfies the recommendation to include at least one open model for transparent evaluation [31]. These models were selected among instruction-tuned, code-pretrained LLMs that matched our criteria, and were also included in recent leaderboards such as Chatbot Arena [39] and the evaluation by Liu et al. [40]. The criteria

ensured feasibility in terms of model size (≈ 7B parameters), availability of chat templates, and suitability for code-related tasks. We limited our evaluation to three models due to the extensive inference cost of processing nearly 18,000 test files, which made scaling to a larger set of models infeasible within available resources.

a) Prompt Generation: To generate prompts for our experiments, we used chat templates,[1] as selected LLMs support structured chat-based interactions. Similarly to previous work using structured prompt templates for classification tasks [27], [29], [35], we designed our prompts by defining a system message to instruct the model, followed by user queries presenting the test cases. The prompt structure was designed for both zero-shot and few-shot experiments, ensuring a controlled inference setup.

In the zero-shot setting, the prompt consists of a structured query instructing the LLM to classify whether a given source code file contains a performance bug. This structured query embeds the test Java source code file as a code snippet. Listing 1 presents the chat template used for the zero-shot prompt.

We extended the prompt structure for the few-shot setting by incorporating two additional Java source code files as labeled examples, one containing a performance bug as a positive example and one without any performance bug as a negative example, as shown in Listing 2. These examples were selected from the same project as the test file to maintain project-level context during inference. The LLM was first presented with these labeled examples, followed by the test code snippet for classification. This configuration corresponds to a two-shot setting, as the LLM is provided with two labeled examples (one buggy and one non-buggy) before classifying the test file. We use the term "few-shot" in this study to align with the broader convention of providing limited labeled examples for guidance.

In our setup, we intentionally did not embed an explicit definition of performance bug in the system message. This isolates the model pre-training knowledge and avoids prompt-induced bias while also preserving token budget, since many files were already near the context limit and adding boilerplate text would have resulted in dropping additional files. In the early phases of our study, we experimented with multiple prompt variants to identify a structure that would reliably support binary classification. Since our task requires a simple "Yes" or "No" output, we used the format `performance bug: <YES or NO>`, inspired by structured classification prompts in [29], which enforces this response format and helped produce less verbose and less ambiguous answers. In line with prior work [1], we restricted the prompt to the scope of a single Java class. This allowed a controlled evaluation at the file level, consistent with the labeling in the original dataset.

b) LLM Execution: The selected LLMs were executed using the Hugging Face repository. Inference was performed under zero-shot and few-shot settings, generating binary predictions that indicate the presence or absence of a performance bug. In all our experiments, we set the sampling temperature to 1 with top-p 0.95 and top-k 50, which introduces some randomness while keeping responses short, and the maximum number of generated tokens (`max_new_tokens`) was 10 to keep the response less verbose.

As shown in the prompts in Listing 1 and Listing 2, the models were instructed to respond strictly in the format `<YES or NO>`. The binary predictions were then derived by interpreting the LLM responses. If the response contained `YES`, it was classified as a buggy file, while `NO` indicated a file without bugs. Additionally, we implemented a heuristic-based interpretation mechanism to handle ambiguous responses. If the response contained words such as "potential", "contains", "possible", "has a few", "several", and "small performance", the file was classified as buggy. This ensured that verbose responses still contributed meaningfully to the classification.

Listing 1: Zero-shot Prompt

```
System: You are an AI binary performance bug classifier
    that identifies whether the provided code contains
   a performance bug or not. Provide response only in
   the following format: performance bug: <YES or NO>.
   Do not include anything else in the response.

User: Does the following code snippet contain any
   performance bug?
    <CODE_SNIPPET>

Response:
```

Listing 2: Few-shot Prompt

```
System:You are an AI binary performance bug classifier
   that identifies whether the provided code contains a
    performance bug or not. Provide response only in
   the following format: performance bug: <YES or NO>.
   Do not include anything else in the response.

User:Does the following code snippet contain any
   performance bug?
    <positive_example>
Assistant:performance bug: YES

User:Does the following code snippet contain any
   performance bug?
    <negative_example>
Assistant:performance bug: NO

User:Does the following code snippet contain any
   performance bug?
    <CODE_SNIPPET>

Response:
```

[1]Templates for Chat Models: https://huggingface.co/docs/transformers/v4.35.1/en/chat_templating

However, if the response remained ambiguous without indicating the likelihood of a bug, the file was classified as non-buggy.

*3) **LLM Fine-tuning**:* To explore whether domain adaptation can improve performance bug detection, we fine-tuned Qwen2.5-Coder-7B using the QLoRA technique. Fine-tuning was performed with the Hugging Face TRL SFTTrainer on an NVIDIA A100 (80 GB) GPU, using 4-bit quantization (bitsandbytes, int4) and mixed precision training. The dataset was split into 40,096 training files ($\approx 60\%$), 3,022 validation files ($\approx 10\%$), and 17,974 testing files ($\approx 30\%$). The fine-tuning process ran for ≈ 19 hours with early stopping; loss decreased from 0.88 to 0.36 in the first epoch and stabilized at 0.44. We then evaluated the fine-tuned model on the test set using the same classification setup as in RQ_1 and RQ_2, focusing on the zero-shot configuration.

*4) **Machine Learning Prediction**:* The machine learning prediction phase involves evaluating traditional ML models to predict performance bugs using the same dataset. Unlike LLMs, these models rely on extracted code and process metrics for classification.

The selection of ML models was guided by prior research on performance bug prediction by Zhao et al. [1]. They used ML classifiers and showed their effectiveness in detecting performance bugs. We evaluated six classifiers, including Random Forest (RF), Decision Tree (DT), Logistic Regression (LR), Support Vector Machines (SVM), Complement Naive Bayes (CNB), and Multi-Layer Perceptron (MLP).

a) Feature Extraction: The ML models were trained using a set of predefined code and process metrics, as proposed in [1]. These metrics capture various characteristics of the source code, such as file complexity, historical changes, and performance-related structural properties. Highly correlated features ($correlation > 0.7$) were removed to prevent redundancy and improve generalization. The dataset for each project was processed to extract these features, which were then used as input for training the ML models.

b) Model Training and Evaluation: To ensure a fair comparison, the ML models were trained using the training split and evaluated on the test split, maintaining consistency with the LLM inference phase. Standard evaluation metrics were used to assess model performance, including precision, recall, and F1 score. We use F1 as the primary measure, and also report precision and recall where relevant. The ML models were evaluated using the complete set of features, including anti-pattern metrics, as Zhao et al. [1] show that their inclusion leads to better predictive performance. We reused scripts in the replication package [41] to obtain results for machine learning models detecting performance bugs, allowing a direct comparison between LLMs and their ML approach. Statistical significance tests were conducted to compare ML-based and LLM-based predictions for performance differences.

III. Results Discussion

In this section, we present and discuss the results of our analysis.

RQ$_1$: *How accurate are LLMs in detecting performance bugs in a zero-shot setting?*

***Motivation**:* We evaluate the baseline capabilities of LLMs to detect performance bugs using zero-shot prompts. Zero-shot evaluation demonstrates the pre-trained knowledge of LLMs to process source code directly, without requiring labeled examples. This provides initial insights into the effectivenss of applying LLMs for performance bug detection, setting a basis for further exploration of their suitability in this domain.

***Approach**:* To address this research question, we evaluated the performance of the selected LLMs in detecting performance bugs at the file level using zero-shot prompts as a baseline. In this setup, the models inferred results directly from raw source code without any prior exposure to labeled examples. This configuration allowed us to assess the inherent ability of pre-trained LLMs to identify performance issues solely based on source code content.

TABLE I: Zero-shot performance metrics of LLMs for detecting performance bugs

Model	Comments Included	Precision	Recall	F1
artigenz coder	**No**	0.0119	0.1542	**0.0221**
artigenz coder	Yes	0.0096	0.1244	0.0178
codellama	**No**	0.0124	0.1592	**0.0230**
codellama	Yes	0.0118	0.1393	0.0218
qwen coder	No	0.0109	0.2438	0.0209
qwen coder	**Yes**	0.0129	0.2338	**0.0244**

***Results**:* Table I summarizes the zero-shot performance metrics of LLMs for detecting performance bugs. In general, we found that zero-shot configurations produce consistently low F1 scores, ranging from 0.0178 to 0.0244 across all models and configurations. Inclusion or exclusion of comments in the source code had minimal impact on the results, with only slight variations in precision, recall, and F1 scores. The bolded F1 scores in the table indicate the best-performing configuration for each model.

These results highlight the poor performance of LLMs with zero-shot prompts for this task. The precision and recall values further illustrate these trends. Interestingly, the inclusion of comments had mixed effects in all models. For Qwen, comments improved precision and F1 score but slightly reduced recall, while comments led to a small decrease in precision, recall, and F1 scores for CodeLlama and Artigenz.

Poor performance across all metrics indicates that zero-shot prompts are insufficient for reliable performance bug detection. Nonetheless, the following research questions examine whether more advanced configurations, such as few-shot prompting, improve model performance.

Figure 4 presents a comparison of the F1 score with zero-shot prompts across different software projects, considering only configurations where comments were included. The results reveal notable variations in the effectiveness of performance bug detection. Certain projects, such as `grpc-java`

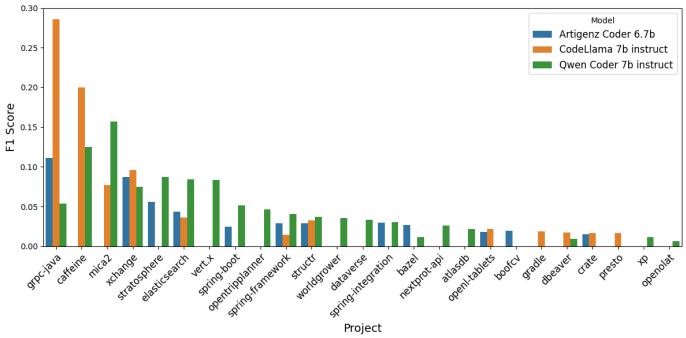

Fig. 4: Zero-shot F1 Score comparison across projects

with CodeLlama ($F1 = 0.2857$) and `mica2` with Qwen ($F1 = 0.1569$), exhibited non-zero detection rates. In contrast, several other projects exhibited an F1 score of zero, indicating a complete failure to identify any performance bugs. This variability indicates project-specific characteristics or patterns that may influence the effectiveness of LLMs in detecting performance bugs with zero-shot prompts.

Answer to RQ$_1$: The zero-shot configurations of LLMs show poor performance in detecting performance bugs, with F1 scores below 0.025 in all models. Inclusion or exclusion of source code comments has minimal impact on performance, resulting in only slight variations in precision, recall, and F1 scores, where differences are small and not practically meaningful. These results highlight the limitations of LLMs in detecting performance bugs without additional context or fine-tuning.

RQ$_2$:*What is the impact of using few-shot prompts on the effectiveness of LLMs in detecting performance bugs?*

***Motivation**:* Few-shot prompting provides the help of labeled examples to an LLM with a limited context. This enables the model to infer patterns and apply them to new instances. We aim to explore whether providing few-shot examples improves the ability of LLMs to detect performance bugs compared to zero-shot setting. Additionally, we investigate the influence of source code comments in few-shot configurations.

TABLE II: Few-shot performance metrics of LLMs for detecting performance bugs with differences relative to zero-shot performance

Model	Comments Included	Precision	Recall	F1
artigenz coder	No	0.0141	0.3831	0.0272 (+23%)
artigenz coder	**Yes**	0.0143	0.3980	**0.0277** (+56%)
codeLlama	No	0.0134	0.3134	0.0258 (+12%)
codeLlama	**Yes**	0.0136	0.2886	**0.0259** (+19%)
qwen coder	No	0.0172	0.6119	0.0335 (+60%)
qwen coder	**Yes**	0.0184	0.5970	**0.0357** (+46%)

***Approach**:* We evaluated the same LLMs adopted for RQ_1 under few-shot prompts. For each model, we provided one labeled buggy and one non-buggy file during inference. This setup should have enabled the models to infer patterns from the provided examples and apply them to new, unseen instances. Similarly to the zero-shot evaluation, we evaluated two configurations: with and without source code comments.

***Results**:* Table II presents the few-shot performance metrics (precision, recall, and F1 score) of the evaluated LLMs. Across all models, F1 scores remain very low, ranging from 0.0258 to 0.0357, with only marginal differences compared to zero-shot configurations. The differences between configurations with and without comments are also minimal, further highlighting the limited impact of a few-shot prompting. The bolded F1 scores in the table represent the best-performing configuration for each model, though these scores still remain well below acceptable levels.

Including comments in the source code led to minor variations in performance between models. For Qwen and Artigenz, the comments slightly improved the precision, although for Qwen this came at the cost of a small reduction in recall. Similarly, CodeLlama showed a slight increase in F1 score and precision, but experienced a minor drop in recall. Overall, the performance of LLMs in detecting performance bugs remains poor, even with few-shot prompting and the inclusion of comments. The project-level results followed the same pattern as in the zero-shot prompt in RQ_1, showing inconsistent and often negligible improvements across systems.

To further examine whether the observed results vary across different types of performance bugs, we conducted a post-hoc analysis on the subset of the dataset that had been annotated with bug categories (e.g., memory leak, infinite loop) by Zhao et al. [1] as part of their original work. From the 340 files with manual labels, 125 could be retrieved and processed in our few-shot setting with Qwen. Although the sample is limited, the analysis provides preliminary insights. As shown in Table III, the recall varied across bug types. Infinite loop bugs achieved a recall of 69%, while memory leaks were correctly detected in 56% of the cases. Performance regression issues, which represented the largest portion of the labeled subset, were classified with 64% recall. Other categories such as stuck, deadlock, and rendering-related regressions reached perfect recall, but these results are based on very few samples. Overall, these findings indicate that model performance may

TABLE III: Post-hoc error analysis of Qwen predictions by performance bug type (few-shot setting)

Bug Type	Samples	Correct Predictions	Recall (%)
Performance regression	84	54	64.29
Infinite loop	13	9	69.23
Not a performance bug	11	6	54.55
Memory leak	9	5	55.56
Stuck	3	3	100.00
Deadlock	2	2	100.00
Rendering regression	2	2	100.00

depend on the type of performance bug, suggesting that certain types of bugs are more difficult for LLMs to identify. This motivates future work on more systematic analyses of these aspects.

Answer to RQ$_2$: Few-shot prompting shows minimal impact on the performance of LLMs in detecting performance bugs, with F1 scores remaining consistently low across both zero-shot and few-shot configurations. Although including comments in the few-shot setup results in infinitesimal variations in F1 scores, these differences are negligible, and overall performance remains unsatisfactory.

RQ$_3$:*What is the impact of using supervised fine-tuning on the effectiveness of LLMs in detecting performance bugs?*

Motivation: Zero-shot and few-shot prompting configurations showed low performance consistently across all evaluated models. Prior research has suggested that supervised fine-tuning may adapt LLMs to specialized tasks and improve effectiveness [42]. Therefore, we investigate whether fine-tuning can enhance the ability of LLMs to detect performance bugs.

Approach: As described in Section II, we fine-tuned Qwen2.5-Coder-7B using QLoRA and evaluated it on the same dataset under the zero-shot configuration.

Results: The fine-tuned `Qwen2.5-Coder-7B` achieved precision 0.022, recall 0.254, and F1 0.041, which represents a +66% relative improvement over its prompt-based zero-shot configuration with comments (F1 0.024). Compared to prompt-based configurations, this gain is modest, but measurable. The improvement is primarily due to higher recall, suggesting that fine-tuning increased the model sensitivity to buggy cases at the cost of lower precision. However, the absolute scores remain very low, highlighting the difficulty of detecting performance bugs and the limited effect of adapting a single model to this dataset.

TABLE IV: Fine-tuning performance metrics of Qwen

Model	Precision	Recall	F1
Qwen2.5-Coder-7B (Fine-tuned)	0.022	0.254	0.041 (+66%)

Answer to RQ$_3$: Supervised fine-tuning slightly improves detection compared to prompt-only settings, but overall performance remains limited, indicating that broader or more diverse fine-tuning data, or more advanced adaptation strategies, may be required to achieve practical effectiveness.

RQ$_4$:*How do LLMs compare to traditional machine learning models in detecting performance bugs?*

Motivation: The earlier research questions showed that LLMs achieved limited accuracy in detecting performance bugs. This research question aims to relate their performance by comparing them with traditional machine learning models.

TABLE V: Results of the performance comparison between LLMs and ML models

LLM ML Algo	**artigenz**	**codellama**	**qwen**
CNB	0.807 (–)	1.0 (–)	1.0 (–)
DT	0.765 (–)	0.764 (–)	1.0 (–)
LR	0.183 (–)	0.359 (–)	1.0 (–)
MLP	**0.001** (**0.84**)	**0.001** (**0.84**)	**0** (**0.94**)
RF	**0.018** (**0.81**)	**0.018** (**0.81**)	**0.002** (**0.9**)
SVM	0.134 (–)	0.546 (–)	1.0 (–)

*Each cell is formatted as $p - value(\hat{A}_{12})$. Bold p-values indicate statistical significance after Bonferroni correction [43]($p < 0.05$). Bold $\hat{A}_{12}$ values indicate a likelihood of LLM outperforming the ML model. $\hat{A}_{12}$ values are omitted for comparisons where the adjusted p-value is greater than 0.05, as the difference is not statistically significant.

Previous work by Zhao et al. [1] applied ML models to predict performance bugs by leveraging code, process, and performance-specific metrics extracted from source code and historical data. However, these models are constrained by feature engineering and imbalanced datasets, where performance bugs are rare (e.g., only 0.84% of files contain performance bugs in the dataset curated by [1]). In contrast, LLMs process code as text and infer patterns without explicit code metric extraction. This motivates our investigation into how LLMs compare against traditional ML models in terms of bug prediction performance. This comparison highlights differences in modeling assumptions and the ability of ML and LLMs to capture performance-related patterns.

Approach: To compare the performance of LLMs with traditional ML models, we evaluated the top-performing configurations of the selected LLMs against the ML baselines described in Section II-4. For a fair comparison, we compared the best-performing LLM configurations based on F1 scores from our zero-shot and few-shot experiments with the ML models trained on the same dataset.

As shown in Table II, LLMs achieved their highest F1 scores (in bold) when using few-shot prompts and including comments in the source code. To ensure consistency in evaluation, we reused the same dataset and experimental protocol described in Section II-4, enabling a comparison of F1 scores between LLMs and ML models.

Thereafter, to assess whether an LLM outperforms an ML model, we performed an analysis using the Wilcoxon signed rank test [33] for each pair of <LLM, ML model> to compare their respective project-wise F1 scores. We set the significance level at 0.05, which means that differences with p-values below this threshold are considered statistically significant. To account for multiple comparisons, p-values were adjusted using the Bonferroni correction [43] across the 18 pairwise tests. In addition to the Wilcoxon signed-rank test, we use Vargha and Delaney $\hat{A}_{12}$ effect size, to assess the magnitude of observed differences. Given two related paired samples X

and Y (for example, F1 scores from an LLM and an ML model), the effect size of $\hat{A}_{12}$ represents the proportion of pairs where X (LLM) is higher than Y (ML model). The value of $\hat{A}_{12}$ ranges from 0 to 1 and is interpreted using the thresholds provided in [44]. An $\hat{A}_{12}$ value of 0.5 suggests that there is no significant difference in the F1 score between the LLM and ML model. A value larger than 0.5 indicates that the LLM is more likely to outperform the ML model, while a value lower than 0.5 suggests the opposite.

Results: Table VI summarizes the performance of the traditional ML baselines. As shown, all models achieve very low F1-scores, confirming that classical learning approaches also struggle to detect performance bugs under severe class imbalance.

When comparing to the few-shot performance of the evaluated LLMs using their best configuration from Table II (Artigenz = 0.0277, CodeLlama = 0.0259, and Qwen = 0.0357), the LLMs show marginal improvements only over the weakest ML baselines (MLP and RF) while performing below the remaining models. This highlights that both model families struggle while detecting performance bugs

To understand the significance of these differences, Table V reports the adjusted *p-values* ($\hat{A}_{12}$) for pairwise comparisons between LLMs and ML models. For the comparisons that remain statistically significant after Bonferroni correction, $\hat{A}_{12}$ exceeds 0.8, indicating that the LLMs are more likely to outperform MLP and Random Forest. For all other ML models, the adjusted *p-values* are greater than 0.05, suggesting lack of statistically significant differences.

For Artigenz, after Bonferroni correction only the comparisons against MLP (0.84) and Random Forest (0.81) remain significant, indicating a higher probability of outperforming these two ML models. For all other ML models, the adjusted *p-values* are greater than 0.05, so no significant differences are observed. Similarly, CodeLlama shows significant advantages only against MLP (0.84) and Random Forest (0.81), while for the remaining ML models, no significant differences are found. For Qwen, after Bonferroni correction only the comparisons against MLP ($\hat{A}_{12} = 0.94$, adjusted $p = 0.000$) and Random Forest ($\hat{A}_{12} = 0.90$, adjusted $p = 0.002$) remain significant. For CNB, DT, LR, and SVM, the adjusted *p-values* are greater than 0.05, so no significant differences are observed.

Overall, the results highlight the limitations of LLMs in outperforming traditional ML models for performance bug detection. While LLMs performed better than ML models in some corner cases (e.g., against MLP and RF), they can be considered exceptions rather than a consistent trend, which should be read in light of the extreme class imbalance in our dataset. For the remaining ML models, the *p-values* are greater than 0.05, showing no significant differences. The poor performance of LLMs in most comparisons highlights the challenges of relying solely on pre-trained knowledge and zero-shot or few-shot configurations for performance bug detection.

TABLE VI: ML models performance for detecting performance bugs

	MLP	RF	CNB	LR	SVM	DT
F1	0.001	0.019	0.049	0.069	0.070	0.074

Answer to RQ$_4$: LLMs do not consistently outperform traditional ML models in detecting performance bugs. Non significant performance differences are observed for all models except MLP and Random Forest. Therefore, relying only on pre-trained knowledge with zero-shot or few-shot prompting is not sufficient for dependable performance bug detection, and traditional ML models remain competitive.

IV. Implications

This section discusses the implications of our findings for practitioners and researchers.

For practitioners. Our study shows that LLMs currently struggle to reliably detect performance bugs in software code, even when provided with positive and negative examples through few-shot prompting or supervised fine-tuning. It is worth noting that our study involved LLMs with smaller parameter sizes than those of frontier models. Therefore, it remains to be demonstrated whether the latter can achieve higher effectiveness in performance bug detection. Nonetheless, our study provides preliminary insights suggesting that LLMs with around 7B parameters are not suitable as standalone tools for performance bug detection in practical settings. Although supervised fine-tuning can slightly enhance their bug detection capabilities, the improvements are negligible. Moreover, fine-tuning is computationally expensive and requires substantial resources, making it a challenging approach for practical adoption. Therefore, practitioners should refrain from attempting to deploy or fine-tune such LLMs for performance bug detection and should instead consider experimenting with larger models, or alternatively, use traditional static analysis tools that can detect specific types of performance bugs more effectively.

Besides, LLMs may still provide value to developers in other ways related to software performance assurance. For example, practitioners could leverage them to support performance debugging activities, such as explaining why a particular file exhibits performance issues or highlighting potential inefficiencies and problematic code patterns. Even if their judgments are unreliable in isolation, such explanations could help developers prioritize code inspections and better understand recurring performance anti-patterns. Furthermore, LLMs could assist in educational or documentation contexts by providing natural-language explanations of performance issues [45]. These potential roles extend beyond pure classification and suggest that LLMs still hold promise for augmenting developer workflows in diagnosing and resolving performance bugs. Nonetheless, these avenues must be empirically validated

to assess LLMs capabilities for these tasks. We encourage future work to explore these directions.

For researchers. From a research perspective, our findings align with previous studies showing that LLMs face challenges in tasks related to software analysis that require deep reasoning [28]. Although LLMs have shown promise in areas such as syntactic bug detection and test case generation [24], their effectiveness in detecting performance-related issues remains limited. Our results suggest that LLMs struggle to infer complex execution behaviors, which is crucial for performance bug detection. This aligns with the findings of LLM-based security analysis [27], where models often fail to generalize beyond simple patterns and require substantial fine-tuning to be useful.

A promising direction is the design of hybrid approaches. Traditional static analysis tools are effective in detecting specific types of performance bugs, such as issues related to loops or conditionals [21], [46], while LLMs can offer more general reasoning and explanations in natural language. Combining these strengths could reduce false positives and improve recall by letting static tools surface candidates while LLMs generate concise justifications or remedial suggestions. Moreover, our setup provided the LLM only with single-class (Java class) context, whereas some performance bugs may require repository-level information or cross-class interactions to become apparent. Hence, it shall therefore be worth to investigate which levels of context (file, package, or repository level) most effectively improve LLM reasoning in performance bug detection. In our study, it is also possible that the limited scope of examples in the few-shot prompts was insufficient to guide the models toward generalizing performance-related patterns. Moreover, the evaluated models may lack the capacity to capture deeper runtime behavior beyond surface-level textual patterns, which is often essential in identifying performance issues.

V. Threats to validity

a) ***Construct validity****:* Our study evaluates the ability of LLMs to detect performance bugs by analyzing source code as plain text. Although this aligns with our research goal, it does not consider whether structural information, such as abstract syntax trees or control flow graphs, could improve LLM performance in this task. Future work should investigate whether alternative representations or hybrid approaches improve LLM-based bug detection.

Additionally, our evaluation focuses on zero-shot and few-shot prompting for RQ_1 and RQ_2, which means that, in these settings, we did not fine-tune the models on performance bug data. Although this reflects real-world usage scenarios in which fine-tuning may not be practical, it also means that the LLMs evaluated under these configurations were not explicitly optimized for this task. While we also investigate supervised fine-tuning for a selected model as part of our study, future work could further explore whether fine-tuning additional models improves detection accuracy, and how it compares to approaches solely based on prompt engineering techniques.

The available dataset [41] provides limited fine-grained labels of performance bug categories. However, preliminary checks on a small labeled subset suggested variability by bug type, a direction we leave for future work.

Finally, we note that experiments were run on an NVIDIA A30 (24 GB); larger memory size with longer context capacity may affect the ability to process longer files or additional tokens, potentially affecting outcomes modestly.

b) ***External validity****:* Our findings are based on selected 31 open-source Java projects from GitHub, which may not fully represent proprietary or large-scale enterprise software. However, these projects span diverse domains and codebases, making them a reasonable sample for investigating LLM performance on real-world software. Future research shall explore the performance of LLMs in closed-source systems and large-scale industrial applications.

c) ***Internal validity****:* The experimental setup was designed to minimize biases. We standardized the prompting techniques, ensuring all models were tested under consistent zero-shot and few-shot prompts. However, both prompt design and inference parameters can introduce subjectivity into LLM outputs. We experimented with multiple prompt formulations before finalizing the structure used in this study, highlighting the sensitivity of results to prompt design. In addition, inference was performed with temperature 1, top-p 0.95, and top-k 50, which introduces nondeterminism but balances reproducibility and exploration. Exploring alternative prompt formulations, different temperature values, or other decoding strategies could be an interesting extension for future work.

Moreover, LLM performance is inherently non-deterministic, meaning slight variations in execution may produce different outputs. To mitigate this, we used identical decoding settings across all models to minimize variability. Despite these considerations, LLM-based predictions remain sensitive to randomness, and the results may vary slightly between different inferences. We did not perform repeated runs per configuration due to computational cost, which limits our assessment of variability across runs.

Additionally, a potential limitation of our study arises from the fact that we employed the smallest models available, all in the 7B parameter range, as running additional models would have required high computational resources given the dataset size. Consequently, we had to exclude files that were larger than the context size of any of the models used. This may have influenced our results, as a larger set of files could provide a more comprehensive evaluation. Future work could explore ways to address this limitation, such as employing models capable of handling larger inputs or adopting techniques to process long files more effectively.

VI. Related work

a) ***Performance Bug Detection****:* Performance bug detection has been tackled by machine learning models trained

on static and process-based code metrics. Prior studies have developed feature-based defect prediction techniques, extracting features such as code complexity, historical bug reports, and execution patterns to classify files as buggy or non-buggy. For example, [1] proposed models that employ performance-specific metrics to improve defect prediction. Beyond the fact that this approach reveals very limited defect prediction capabilities, all these approaches require manual feature engineering, making them resource-intensive and limiting their generalizability across diverse software projects.

An alternative approach involves detecting performance anti-patterns, which are coding styles associated with performance inefficiencies. Li et al. [47] explored rule-based techniques to identify performance bugs, but these methods rely on hand-crafted rules that may not cover all possible performance issues. Although these approaches have achieved high accuracy in specific cases, they are difficult to generalize and struggle to capture dynamic behaviors, which are essential for identifying performance bottlenecks.

Despite progress in ML-based detection, these methods remain constrained by feature engineering requirements and limited generalizability. Our work differs from these approaches by evaluating LLMs as a potential alternative for performance bug detection, examining their ability to process raw source code as text without predefined feature sets.

*b) **LLMs and Software Performance**:* A recent line of work has explored the use of LLMs to optimize software performance. In a recent paper, Shypula *et al.* [48] curated a dataset of performance-improving edits made by human programmers across over 77,000 competitive C++ programming submission pairs, called PIE4Perf. In the same paper, they used PIE4Perf to propose a broad range of adaptation strategies for code optimization, including prompting and fine-tuning strategies. Gao *et al.* [49] proposed SBLLM, a search-based LLM framework that enables iterative refinement and discovery of improved optimization methods. SBLLM synergistically integrates LLMs with evolutionary search for code optimization. Garg *et al.* [50] presented DeepDev-PERF, a transformer-based approach to suggest performance improvements for C# applications. Di Menna *et al.* [51] investigated how incorporating code execution information into language models affects their ability to optimize code. Yi *et al.* [52] examined whether LLMs can generate fast code using a dataset of 65 real-world tasks mined from open-source Java programs.

Another body of research aims to improve the efficiency of language models for code through various compression strategies [53]–[56]. Wei *et al.* [55] empirically evaluate quantized models on code generation tasks across different dimensions: (i) resource usage and carbon footprint, (ii) accuracy, and (iii) robustness. Panichella [53] proposes Morph, a method that combines metamorphic testing with many-objective optimization for a robust distillation of LLMs for code. Shi *et al.* [54] propose Compressor, a novel approach that can compress the pre-trained models of code into extremely small models with negligible performance sacrifice. D'Aloisio *et al.* [56] empirically investigate the impact of three well-known compression strategies – knowledge distillation, quantization, and pruning – across three different software engineering tasks.

While prior work has focused on improving the efficiency of code using LLMs, or on improving the efficiency of language models for code, our study provides the first empirical evaluation of LLMs for performance bug detection.

*c) **LLM-Based Bug Detection**:* Previous research on LLMs for bug detection and static analysis has largely focused on security vulnerabilities and functional defects, rather than performance-related issues. Zhou et al. [28] found that while LLMs can identify common vulnerability patterns, they struggle with complex, multistep reasoning tasks. Similarly, Dozono et al. [35] have attempted to fine-tune LLMs on defect prediction datasets, but fine-tuning remains computationally expensive and it is not always practical for real-world deployment.

Despite these challenges, there is ongoing research into improving LLM-based defect detection through prompt engineering and pre-trained LLMs. Recent work by Boukhlif et al. [24] compares how prompt engineering strategies are used in LLM-based software testing studies, highlighting its widespread use, although performance gains remain modest in many cases. Curto et al. [25] investigate whether fine-tuned LLMs like Llama 3 and Code Llama can effectively perform static application security testing. Their results show strong performance on benchmark datasets, though the authors highlight task dependency and the need for further validation on unbalanced, real-world data.

While these studies primarily explore LLMs in security and static analysis contexts, our research takes a different direction by evaluating LLMs potential for performance bug detection, a domain where execution behaviors are more challenging to infer from static code.

Unlike prior LLM-based studies that mainly address functional or security vulnerabilities, our work focuses on performance bugs.

VII. Conclusion and future work

This paper presented an empirical study investigating the effectiveness of LLMs for performance bug detection. Despite their success in various software engineering tasks, our results indicate that LLMs struggle to reliably detect performance bugs from raw source code, even when using few-shot prompting or supervised fine-tuning. Our findings suggest that LLMs with approximately 7B parameters are not yet suitable as standalone tools for detecting performance bugs, likely due to their limited capacity for reasoning about execution behavior.

Future research should investigate whether larger and more advanced LLMs can improve the effectiveness of performance bug detection. Another promising direction involves developing hybrid approaches that combine the strengths of traditional static analysis tools with the general-purpose capabilities of LLMs, potentially within an agentic workflow. Similarly, future work could also assess whether combining LLMs with

code metrics-based ML models could enhance performance bug detection.

Although current LLMs are not reliable standalone detectors, they can still augment performance engineering as assistants for explanation, assessment, and documentation, or as rationale generators in hybrid pipelines that integrate static analysis. We encourage future research to further explore these directions.

ACKNOWLEDGMENT

This work is partially supported by: (i) Italian Government (Ministero of University and Research, PRIN 2022 PNRR) – cod. P2022SELA7: "RECHARGE: monitoRing, tEsting, and CHaracterization of performAnce Regressions" – D.D. n. 1205, 28/7/2023; (ii) European Union – NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR), Project:"ICSC - Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing" – cod. CN00000013 – D.D. n. 3138del 16/12/2021; (iii) European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216- 2, KDT232RIA 0001.

REFERENCES

[1] G. Zhao, S. Georgiou, S. Hassan, Y. Zou, D. Truong, and T. Corbin, "Enhancing performance bug prediction using performance code metrics," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 50–62.

[2] 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.

[3] A. Nistor, *Understanding, detecting, and repairing performance bugs*. University of Illinois at Urbana-Champaign, 2014.

[4] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[5] M. A. K. Azad, N. Iqbal, F. Hassan, and P. Roy, "An empirical study of high performance computing (hpc) performance bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 194–206.

[6] 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.

[7] ben manes, "Binarytracereader.java in caffeine," Accessed January 30, 2025 from https://github.com/ben-manes/caffeine/blob/v3.0.5/simulator/src/main/java/com/github/benmanes/caffeine/cache/simulator/parser/BinaryTraceReader.java, n.d.

[8] IQSS, "Index.java in dataverse," Accessed January 30, 2025 from https://github.com/IQSS/dataverse/blob/v5.9/src/main/java/edu/harvard/iq/dataverse/api/Index.java, n.d.

[9] F. I. Vokolos and E. J. Weyuker, "Performance testing of software systems," in *Proceedings of the 1st International Workshop on Software and Performance*, ser. WOSP '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 80–87. [Online]. Available: https://doi.org/10.1145/287318.287337

[10] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa, "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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 188–199. [Online]. Available: https://doi.org/10.1145/3338906.3338912

[11] L. Traini, F. Di Menna, and V. Cortellessa, "Ai-driven java performance testing: Balancing result quality with testing time," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 443–454. [Online]. Available: https://doi.org/10.1145/3691620.3695017

[12] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 409–425. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq

[13] L. Traini, V. Cortellessa, D. Di Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in java software: are we there yet?" *Empirical Software Engineering*, vol. 28, no. 1, p. 13, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x

[14] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 254–265. [Online]. Available: https://doi.org/10.1145/3213846.3213874

[15] 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*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 48–57. [Online]. Available: https://doi.org/10.1145/3661167.3661196

[16] ——, "Is code coverage of performance tests related to source code features? an empirical study on open-source java systems," *Empirical Software Engineering*, vol. 30, no. 6, p. 157, 2025. [Online]. Available: https://doi.org/10.1007/s10664-025-10712-3

[17] 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. [Online]. Available: https://doi.org/10.1145/3377811.3380351

[18] 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

[19] L. Traini, "Exploring performance assurance practices and challenges in agile software development: An ethnographic study," *Empirical Software Engineering*, vol. 27, no. 3, p. 74, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10069-3

[20] 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

[21] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 902–912.

[22] C. Zhang, J. Li, D. Li, and X. Lu, "Understanding and statically detecting synchronization performance bugs in distributed cloud systems," *IEEE Access*, vol. 7, pp. 99 123–99 135, 2019.

[23] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 254–264. [Online]. Available: https://doi.org/10.1145/2338965.2336784

[24] M. Boukhlif, N. Kharmoum, and M. Hanine, "Llms for intelligent software testing: a comparative study," in *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security*, 2024, pp. 1–8.

[25] C. Curto, D. Giordano, D. G. Indelicato, and V. Patatu, "Can a llama be a watchdog? exploring llama 3 and code llama for static application security testing," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2024, pp. 395–400.

[26] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, and Y. Liu, "An empirical study of automated vulnerability localization with large language models," *arXiv preprint arXiv:2404.00287*, 2024.

[27] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" *arXiv preprint arXiv:2403.18624*, 2024.

[28] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[29] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the effectiveness of large language models in detecting security vulnerabilities," *arXiv preprint arXiv:2311.16169*, 2023.

[30] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[31] S. Baltes, F. Angermeir, C. Arora, M. Muñoz Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald *et al.*, "Guidelines for empirical studies in software engineering involving large language models," *arXiv e-prints*, pp. arXiv–2508, 2025.

[32] M. Imran, V. Cortellessa, D. Di Ruscio, R. Rubei, and L. Traini, "An Empirical Investigation of Large Language Models Effectiveness on Performance Bug Detection - Replication Package," 2026. [Online]. Available: https://github.com/imran9pk/replication-package_perf_bugs_detection_LLM

[33] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[35] K. Dozono, T. E. Gasiba, and A. Stocco, "Large language models for secure code assessment: A multi-language empirical study," *arXiv preprint arXiv:2408.06428*, 2024.

[36] Huggingface, "Cdellama Model Card," https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf, 2025, [Online; accessed 05-January-2025].

[37] ——, "Qwen Codel Model Card," https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct, 2025, [Online; accessed 05-January-2025].

[38] ——, "Artigenz Coder Model Card," https://huggingface.co/Artigenz/Artigenz-Coder-DS-6.7B, 2025, [Online; accessed 05-January-2025].

[39] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. Jordan, J. E. Gonzalez *et al.*, "Chatbot arena: An open platform for evaluating llms by human preference," in *Forty-first International Conference on Machine Learning*, 2024.

[40] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[41] NintyFive, "Enhancing Performance Bug Prediction Using Performance Code Metrics-Replication Package," 2024. [Online]. Available: https://github.com/NintyFive/MSR2024-Replication-Package

[42] F. V. Ruiz, M. Hort, and L. Moonen, "The art of repair: Optimizing iterative program repair with instruction-tuned models," *arXiv preprint arXiv:2505.02931*, 2025.

[43] O. J. Dunn, "Multiple comparisons among means," *Journal of the American Statistical Association*, vol. 56, no. 293, pp. 52–64, 1961.

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

[45] S. S. Sijwali, A. M. Colom, A. Guo, and S. Saha, "Fixing performance bugs through llm explanations," in *2025 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2025, pp. 102–109.

[46] L. Song and S. Lu, "Performance diagnosis for inefficient loops. in 2017 ieee/acm 39th international conference on software engineering (icse)," 2017.

[47] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[48] A. G. Shypula, A. Madaan, Y. Zeng, U. Alon, J. R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=ix7rLVHXyY

[49] S. Gao, C. Gao, W. Gu, and M. R. Lyu, "Search-based llms for code optimization," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 578–590.

[50] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "Deepdev-perf: a deep learning-based approach for improving software performance," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 948–958. [Online]. Available: https://doi.org/10.1145/3540250.3549096

[51] F. Di Menna, L. Traini, G. Bavota, and V. Cortellessa, "Investigating execution-aware language models for code optimization," in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, 2025, pp. 204–215.

[52] L. Yi, G. Gay, and P. Leitner, "An experimental study of real-life llm-proposed performance improvements," 2025. [Online]. Available: https://arxiv.org/abs/2510.15494

[53] A. Panichella, "Metamorphic-based many-objective distillation of llms for code-related tasks," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1001–1013.

[54] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 mb," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556964

[55] X. Wei, S. K. Gonugondla, S. Wang, W. Ahmad, B. Ray, H. Qian, X. Li, V. Kumar, Z. Wang, Y. Tian, Q. Sun, B. Athiwaratkun, M. Shang, M. K. Ramanathan, P. Bhatia, and B. Xiang, "Towards greener yet powerful code generation via quantization: An empirical study," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 224–236. [Online]. Available: https://doi.org/10.1145/3611643.3616302

[56] G. d'Aloisio, L. Traini, F. Sarro, and A. Di Marco, "On the compression of language models for code: An empirical study on codebert," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2025, pp. 12–23.