AI-driven Java Performance Testing: Balancing Result Quality with Testing Time*

Luca Traini
luca.traini@univaq.it
University of L'Aquila
Italy

Federico Di Menna
federico.dimenna@graduate.univaq.it
University of L'Aquila
Italy

Vittorio Cortellessa
vittorio.cortellessa@univaq.it
University of L'Aquila
Italy

ABSTRACT

Performance testing aims at uncovering efficiency issues of software systems. In order to be both effective and practical, the design of a performance test must achieve a reasonable trade-off between result quality and testing time. This becomes particularly challenging in Java context, where the software undergoes a warm-up phase of execution, due to just-in-time compilation. During this phase, performance measurements are subject to severe fluctuations, which may adversely affect quality of performance test results. Both practitioners and researchers have proposed approaches to mitigate this issue. Practitioners typically rely on a fixed number of iterated executions that are used to warm-up the software before starting to collect performance measurements (*state-of-practice*). Researchers have developed techniques that can dynamically stop warm-up iterations at runtime (*state-of-the-art*). However, these approaches often provide suboptimal estimates of the warm-up phase, resulting in either insufficient or excessive warm-up iterations, which may degrade result quality or increase testing time. There is still a lack of consensus on how to properly address this problem. Here, we propose and study an AI-based framework to dynamically halt warm-up iterations at runtime. Specifically, our framework leverages recent advances in AI for Time Series Classification (TSC) to predict the end of the warm-up phase during test execution. We conduct experiments by training three different TSC models on half a million of measurement segments obtained from JMH microbenchmark executions. We find that our framework significantly improves the accuracy of the warm-up estimates provided by *state-of-practice* and *state-of-the-art* methods. This higher estimation accuracy results in a net improvement in either result quality or testing time for up to +35.3% of the microbenchmarks. Our study highlights that integrating AI to dynamically estimate the end of the warm-up phase can enhance the cost-effectiveness of Java performance testing.

KEYWORDS

Microbenchmarking, JMH, Java, Time Series Classification

1 INTRODUCTION

Software performance is a critical non-functional aspect of software systems. Technology organizations use performance testing to uncover performance bugs that might deteriorate software efficiency. Nevertheless, performance testing requires careful design in order to be effective. A significant challenge is to design an adequate number of execution repetitions that mitigate the variability of performance measurements [12, 44, 46]. This typically involves balancing the need for quality of performance test results against the practical constraints of resources and testing time [2, 3, 10, 23, 32, 37].

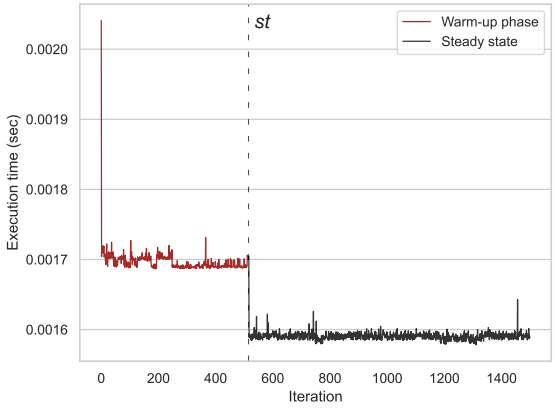

Figure 1: Execution time of a Java microbenchmark (from the *Roaring Bitmap* project) over consecutive iterations. The execution is characterized by an initial warm-up phase and a subsequent steady-state of performance. The grey dotted line indicates the iteration *st* at which steady-state is attained.

Finding this balance becomes particularly difficult in the Java context, where software execution undergoes an initial warm-up phase due to just-in-time compilation [6]. During this phase, the Java Virtual Machine (JVM) performs a wide range of optimizations, leading to fluctuations in performance behavior (as shown in Fig. 1) that may adversely affect result quality [22, 55]. To mitigate this issue, it is common practice to conduct a number of "*warm-up iterations*," with the sole goal of warming up the JVM, before starting to collect performance measurements.

An insufficient number of warm-up iterations might compromise the results quality, thus misleading performance evaluation. On the other hand, unnecessary warm-up iterations increase testing time, thus hindering the adoption of performance testing in practice [18, 30, 54]. An accurate estimation of the warm-up phase is paramount to achieve cost-effective performance testing.

Software engineers typically defines a fixed number of *warm-up iterations* based on their domain expertise [37]. However, studies show that using a fixed number of iterations may misrepresent the warm-up phase [22, 37, 55]. In response to this, researchers developed state-of-art techniques that leverage statistical heuristics

*This article has been accepted for publication in *The 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*.
This version of the manuscript is a preprint and may differ from the final published version in terms of content and formatting.
Citation information: DOI 10.1145/3691620.3695017

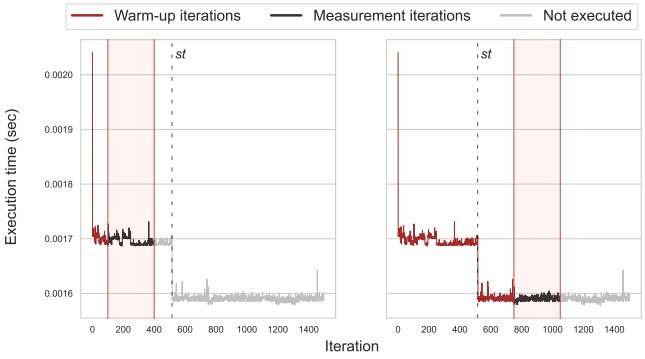

Figure 2: The left box shows an example of an insufficient number of warm-up iterations, which leads to the collection of measurements unrepresentative of the steady-state. The right plot depicts a scenario with an excessive number of warm-up iterations, which increases the testing time.

to dynamically stop the number of warm-up iterations at run-time [22, 37]. Despite their advantages, these techniques are prone to significant inaccuracies that may either compromise result quality or increase testing time [55]. It still remains unclear how to effectively stop warm-up iterations to ensure result quality in a timely manner.

In this paper, we propose an AI-based framework to dynamically stop warm-up iterations at run-time. Our framework utilizes Time Series Classification (TSC) models to predict whether a set of measurements is stable or not, and it exploits this capability to dynamically determine the end of the warm-up phase. We integrate our framework with three state-of-the-art TSC models, and conduct experiments on JMH microbenchmarks, a form of performance testing commonly used in Java software [40]. Results show that our framework noticeably improves the cost-effectiveness of performance testing.

The main contributions of our paper are summarized as follows:

- We propose an AI-based framework that can integrate different TSC models to improve the cost-effectiveness of performance testing.
- We perform a comprehensive evaluation of our framework. The results show that, compared to both the state-of-practice and the state-of-the-art, our framework achieves a net improvement—in either result quality or testing time—in up to +27% and +35.3% of the microbenchmarks, respectively.
- We make publicly available the source code of our framework and the pretrained TSC models to aid future studies[1].

[1]Our replication package, including the dataset, source code and pre-trained TSC models, is publicly available at https://doi.org/10.5281/zenodo.13749258.

```
@Warmup(iterations = 5)
@Measurement(iterations = 10)
public class StringSerializerBenchmark
{
    ...
    @Benchmark
    public byte[] builtInSerializer()
    {
        return s.getBytes("UTF-8");
    }
    ...
}
```

Listing 1: Example of a JMH microbenchmark from the *protostuff* Java library. The `@Warmup` annotation is used to define the number of warm-up iterations.

2 BACKGROUND

In this section, we discuss the background related to our study, including Java microbenchmarking and Time Series Classification.

2.1 Java Microbenchmarking

Microbenchmarking is a type of unit-level performance testing, commonly used in Java context [40]. A microbenchmark repeatedly executes a small portion of code, such as a Java method, while gathering measurements of its execution time. Due to just-in-time compilation, microbenchmarks are subject to performance variability in the first phase of their execution,[2] also known as *warm-up*. During this phase, the JVM detects frequently executed loops or methods, and it dynamically compiles them into optimized machine code. After the completion of the warm-up phase, the microbenchmark is said to be executing at a *steady-state of performance* [6].

Software engineers employ *warm-up iterations* to ensure that the microbenchmark achieves a steady-state before starting to collect measurements. As shown in Figure 2, executing an appropriate number of the warm-up iterations is paramount to achieve cost-effective performance testing. Insufficient number of warm-up iterations may produce measurements that do not reflect the software's true steady-state performance, thus detrimentally affecting result quality. Conversely, excessive warm-up iterations lead to unnecessary executions that prolong the testing time.

State-of-practice (SOP). Practitioners usually predefine a fixed number of warm-up iterations based on their domain expertise. To configure warm-up iterations, they typically rely on Java Microbenchmark Harness (JMH) [48], the de-facto standard for building, configuring, and running Java microbenchmarks. JMH allows to configure the number of warm-up iterations directly in the microbenchmark's source code, using Java annotations, as shown in Listing 1. Nevertheless, prior work has shown that using a fixed

[2]The JVM allows disabling JIT compilation to reduce performance variability. However, doing so would give an unrealistic representation of software performance, as JIT compilation is typically enabled in production environments [47].

number of iterations may often produce suboptimal estimates of the warm-up phase [22, 37, 55].

State-of-the-art (SOTA). Researchers have developed techniques that can dynamically stop warm-up iterations at run-time. Georges *et al.* [22] introduced a first such approach, using a preset threshold on the coefficient of variation [17] to determine the end of the warm-up phase. However, subsequent studies have identified Georges *et al.*'s heuristic as both inaccurate [32] and unrealistic [37, 55] in practical scenarios. Another technique was recently proposed by Laaber *et al.* [37]. Similarly to Georges *et al.*'s heuristic, it leverages statistical metrics to estimate the end of the warm-up phase. At each microbenchmark iteration, this technique checks whether adding more performance measurements are likely to change the distribution of performance measurements, and it uses this information to dynamically stop warm-up iterations at run-time. Laaber *et al.* provide three different variants of this technique, based on distinct statistical metrics, namely coefficient of variation (COV) [17],[3] relative confidence interval width [14, 31] (RCIW), and Kullback-Leibler divergence [23, 36] (KLD). Prior work has shown that Laaber *et al.*'s technique is more accurate than the SOP in estimating the end of the warm-up phase [55].

2.2 Time Series Classification

Time Series Classification (TSC) involves assigning predefined labels to time-ordered sequences of data points based on their characteristics or patterns. TSC problems arise in many domains, such as human activity recognition [8], e-health [50], natural disasters [4], and finance [43]. Due to its broad applicability, hundreds of TSC algorithms have been proposed over the last decades [5, 45]. These algorithms span various categories, including distance-based [42], dictionary-based [51], convolutional-based [15, 52], or deep learning approaches [28]. Recently, the latter two categories have demonstrated notable advancements, enabling them to attain state-of-the-art performance on established TSC benchmarks [19, 28, 45, 53], such as the UCR archive [13]. In this work, we study the efficacy of three state-of-the-art TSC algorithms to dynamically stop warm-up iterations of microbenchmarks. Specifically, we investigate one convolutional-based algorithm, namely ROCKET [15], and two deep learning models, namely FCN [59] and OSCNN [53].

3 METHODOLOGY

In this section, we present the methodology of our AI-based framework. We first introduce an overview of the main phases of our framework and then discuss the details of each phase.

3.1 Overview

As shown in Fig. 3, our framework involves three main phases: *Data Preprocessing*, *Model Training*, and *Application*. The first two phases are executed offline, the last one during performance test execution.

During the *Data Preprocessing* phase, our framework begins by processing a time series of performance measurements with a known warm-up end. This process involves segmenting the time series and labeling each segment as *stable* or *unstable* based on the presence of warm-up measurements. The labeled segments are then used for training the Time Series Classification model (*Model Training*). Finally, the framework leverages the trained model at runtime to dynamically stop the number of warm-up iterations (*Application*).

3.2 Data Preprocessing

This phase aims at preparing the dataset that will be used for supervised learning. To achieve this, our framework starts from a time series of performance measurements, and an annotation that denotes the end of the warm-up phase. The time series represents the observed execution time over sequential iterations of a JMH microbenchmark. The annotation marks the specific iteration (st) in which the steady-state of performance is reached.

In this study, we rely on the notion of steady-state defined by Barrett *et al.* [6]. This notion leverages change point detection, namely PELT [34], to determine statistically significant shifts in the time series, and to identify the end of the warm-up phase. Barrett *et al.*'s approach cannot be used at run-time to determine warm-up, since it requires to first run the microbenchmark for an unrealistically large number of iterations, and, subsequently determine the end of the warm-up phase through post-hoc analysis. However, we aim to leverage this approach to build a solid ground-truth for our AI-based framework.

To construct our dataset, we sample smaller (overlapping) segments of fixed size from the original time series, with each segment representing a contiguous block of performance measurements. We then classify these segments with binary labels:

- *Stable*: The segment consists solely of measurements from the steady-state, meaning all measurements were taken after the steady-state st was reached.
- *Unstable*: The segment includes at least one measurement from the warm-up phase, indicating that some measurements were taken before reaching st.

Fig. 3 illustrates a simplified representation of this process.

Our dataset construction involves multiple time series gathered from various microbenchmarks across different software systems. This process yields a heterogeneous dataset of measurement segments that capture diverse performance patterns from a range of software systems.

3.3 Model training

Our learning goal implies the binary classification of segments of performance measurements. We leverage Time Series Classification algorithms to classify each segment as either *stable* or *unstable*. Specifically, we study three different TSC algorithms, which we discuss below:

Fully Convolutional Network (FCN) was proposed by Wang *et al.* [59] for classifying univariate time series. This neural network architecture acts as feature extractor stacking three convolutional layers [21], each followed by a batch normalization layer [27] and ReLU activation layer [20]. The features are then processed through a global average pooling layer [41] and finally passed into a softmax

[3]Both the approaches of Georges *et al.* [22] and Laaber *et al.* [37] use the CV, but they apply it in different ways. Georges *et al.* use a fixed threshold on CV, whereas Laaber *et al.* verify whether the addition of more measurements changes the CV within a specified threshold.

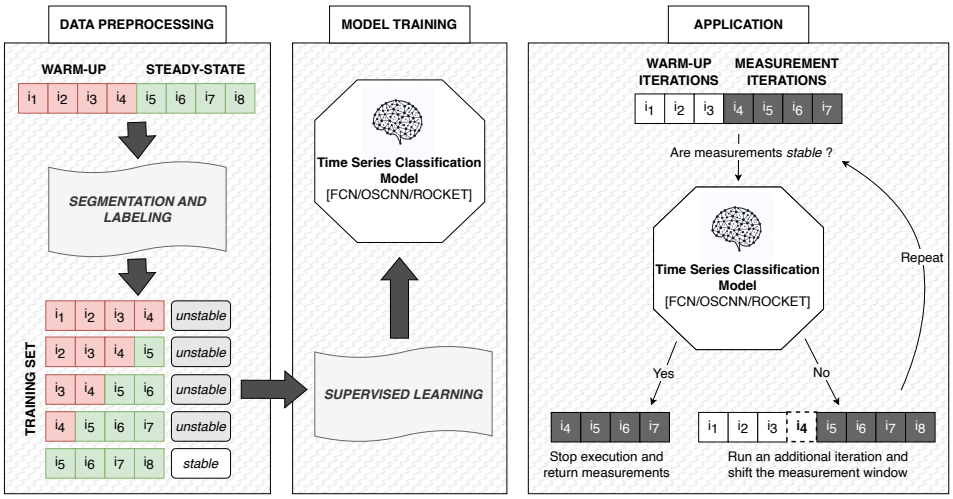

Figure 3: Overview of the main phases of our framework: *Data Preprocessing*, *Model Training* and *Application*.

classifier to obtain the final output label. Prior work has shown that this architecture can achieve state-of-the-art performance in TSC [28].

Omni-Scale Convolutional Network (OSCNN) introduces the Omni-Scale block [53], wherein the kernel sizes for 1-dimensional convolutional neural networks (1D-CNNs) are automatically set through a simple and universal rule. This approach enables capturing an optimal receptive field size, that is a factor that significantly influences the performance of 1D-CNNs for TSC [11].

Random Convolutional Kernel Transform (ROCKET) is a pipeline classifier [15]. It generates a large number of randomly parameterized convolutional kernels and uses these to transform the data through two pooling operations: max value and the proportion of positive values. These two features are concatenated into a feature vector for all kernels. The feature vectors are then used to train a Ridge Classifier [25] using cross-validation. Unlike convolutional neural networks, ROCKET does not involve any hidden layers, thereby providing state-of-the-art performance with limited computational cost [15, 45, 53].

3.4 Application

Our framework employs TSC models to dynamically predict when the microbenchmark reaches a steady-state of performance. As the microbenchmark execution progresses, the framework continuously evaluates the incoming performance measurements using the TSC model. Once the model detects the achievement of the steady-state, the framework promptly halts the microbenchmark execution and returns the current set of measurements for performance evaluation. Fig. 3 presents a snapshot of this procedure using an illustrative scenario. In this scenario, the microbenchmark has completed 3 warm-up iterations, with the current measurement window ranging from the 4th to the 7th iteration. Before proceeding, our framework queries the TSC model, forwarding the current set of measurements to determine if they correspond to a steady-state of execution. If the model predicts that the measurements are *stable*, the microbenchmark execution is halted, and these measurements are returned for performance evaluation. Conversely, if the model classifies them as *unstable*, then the framework makes the test run another iteration, and it shifts the measurement window by one iteration. This process repeats at each iteration until the model identifies a *stable* segment (or until a predefined maximum number of iterations is achieved).

The key insight behind our framework is that if the employed TSC models accurately classify *stable* and *unstable* measurements, then the framework will automatically halt a microbenchmark execution immediately after the warm-up phase ends, thereby providing high-quality performance results with the minimal required testing time.

4 EXPERIMENTAL SETUP

In this section, we outline the experimental procedure, describe the dataset and evaluation metrics employed in our study, and detail the implementation of our framework.

4.1 Dataset of JMH Performance Measurements

We utilize the dataset by Traini *et al.* [55], which includes performance measurements from 586 JMH microbenchmarks across 30 Java software systems (refer to Table 1 for an overview of these systems). To our knowledge, this is the most extensive publicly available dataset of JMH performance measurements. The dataset includes 10 time series of performance measurements per microbenchmark, resulting in a total of 5,860 time series. Each time series comprises performance measurements gathered from 3,000 consecutive microbenchmark iterations performed within a fresh JVM instantiation (often referred to as a *fork* in JMH nomenclature). Each data point in the time series reports the average execution time observed within the microbenchmark iteration. In addition, each time series is annotated with the *st* iteration number at which a steady-state was attained, based on the Barrett *et al.*'s technique [6].

Table 1: Overview of the Java systems, including the name of each GitHub repository, the number of stars and forked repositories, and the number of microbenchmarks involved in the dataset.

Repository	Stars	Forked Repo.	Microbench.
HdrHistogram/HdrHistogram	2,141	251	20
JCTools/JCTools	3,496	554	20
ReactiveX/RxJava	47,702	7,581	20
RoaringBitmap/RoaringBitmap	3,415	535	20
apache/arrow	13,686	3,341	20
apache/camel	5,366	4,896	20
apache/hive	5,370	4,601	20
apache/kafka	27,594	13,571	20
apache/logging-log4j2	3,290	1,559	20
apache/tinkerpop	1,911	786	20
cantaloupe-project/cantaloupe	261	104	19
crate/crate	3,977	546	20
eclipse-vertx/vert.x	14,153	2,043	16
eclipse/eclipse-collections	2,368	581	20
eclipse/jetty.project	3,766	1,899	19
eclipse/rdf4j	347	160	20
h2oai/h2o-3	6,756	1,991	20
hazelcast/hazelcast	5,935	1,803	17
imglib/imglib2	291	93	20
jdbi/jdbi	1,917	333	15
jgrapht/jgrapht	2,535	819	20
netty/netty	32,923	15,763	20
openzipkin/zipkin	16,780	3,069	20
prestodb/presto	15,646	5,265	20
prometheus/client_java	2,134	772	20
protostuff/protostuff	2,016	302	20
r2dbc/r2dbc-h2	196	44	20
raphw/byte-buddy	6,052	776	20
yellowstonegames/SquidLib	447	46	20
zalando/logbook	1,737	257	20

Note that this high number of JMH iterations would be impractical in a real-world scenario due to the extensive testing time required (*e.g.,* the entire data collection required about 93 days according to Traini *et al.* [55]). Nonetheless, we can leverage these time series along with the associated annotation st to build a solid ground-truth for training our TSC models and validate the effectiveness of our framework.

4.2 Experimental procedure

Dataset Preprocessing. The initial phase of our experimental procedure involves generating the dataset for training and evaluating the TSC models. To ensure the effectiveness of the learning process, time series that do not reach a steady-state are excluded, leading to the omission of 10.9% of the time series.

Our dataset construction is based on three main criteria: (i) ensure equal representation of segments from different time series, so no single series dominates the dataset, (ii) achieve a reasonable balance between *stable* and *unstable* segments, and (iii) ensure an adequate coverage of the time series while minimizing measurement redundancy.

To meet the first criterion, we sample 100 segments of 100 measurements each from each time series, following a measurement window similar to that used in prior work [37]. This ensures equal representation of different time series segments in the dataset. To satisfy the second criterion, we sample 50 *stable* and 50 *unstable* segments from each time series, ensuring a balance between the two classes within the dataset. For the third criterion, we use a time series segmentation with adaptive step size to limit measurement redundancy across segments. This strategy is applied independently to both the warm-up and steady-state phases of the time series. Specifically, the segments are selected equidistantly, ensuring that the starting points of the segments are evenly spaced. Specifically, the window segmentation for the 50 *unstable* segments during the warm-up phase uses an adaptive step size of $step = \lfloor (st-1)/50 \rfloor$ where st denotes the iteration where the microbenchmark reaches the steady-state. Similarly, for the steady-state phase, the step size is computed as $step = \lfloor (n-st)/50 \rfloor$, where n denote the length of the time series, namely 3,000. This approach ensures reasonable coverage of the time series while reducing potential measurement redundancy due to overlapping segments.

As a result, we obtain a final dataset of 521,900 measurement segments, including 376,925 (72%) *stable* and 144,975 (28%) *unstable* segments.

Model Training. We adopt a 5-fold cross-validation process for each of the three TSC models. The dataset is divided into five folds of equal size. For each fold, the model is trained on four folds and tested on the remaining one. This process is repeated 5 times, with each fold being used exactly once as the test set. To prevent data leakage, we ensure that measurement segments gathered from the same microbenchmark always appear within the same fold. This guarantees an unbiased evaluation setup, where the model is tested against measurements gathered from unseen microbenchmarks. Furthermore, to increase the representativeness and heterogeneity of each fold, we ensure that each fold has a similar proportion of microbenchmarks per project, using stratified random sampling. The entire cross-validation training process for all the three TSC models required approximately 2 days to complete.

Application. To evaluate our framework, we employ a methodology similar to prior work [37]. Specifically, we mimic the application of the framework during microbenchmark execution through post-hoc analysis. We use a sliding measurement window of 100 performance measurements on the 5,860 time series of measurements from the dataset of JMH performance measurements [55]. For a given time series (*i.e.,* microbenchmark fork), the process begins with a segment containing the first consecutive 100 performance measurements. The framework submits this segment to the TSC model to determine if the measurements are classified as *stable* or not. If the TSC model classifies the measurements as *stable*, then the framework stops the process and returns the measurements M for performance evaluation. Conversely, if the model deems the measurements *unstable*, then the framework shifts the sliding window by one iteration, simulating the execution of an additional warm-up iteration. This process is repeated until the TSC model classifies the measurements as *stable*. Similar to prior work [37], we set an upper limit of 500 warm-up iterations. If this limit is reached, the process stops automatically, and the current set of measurements M is returned for evaluation.

As a result of this process, for each microbenchmark fork, we obtain the number of warm-up iterations estimated by the framework, along with the corresponding set of measurements M provided for performance evaluation.

Similar to the model training phase, when evaluating the framework on a microbenchmark, we consistently use the model trained on the four folds that exclude the target microbenchmark. This approach ensures that our framework is always evaluated against a previously unseen microbenchmark.

4.3 Evaluation metrics

We use different metrics to evaluate the prediction accuracy of TSC models:

- **Precision** represents the ratio of true positive outcomes to the total positive predictions made by the model. In our context, the positives are *stable* measurement segments.
- **Recall** indicates the ratio of true positive outcomes to the total actual positives in the dataset.
- **F1-score** is the harmonic mean of precision and recall, providing a single measure that balances both concerns.
- **Balanced Accuracy** is the average recall obtained across each of the two classes. We use this metric instead of traditional accuracy due to the imbalanced nature of our dataset.

The following metrics are used for evaluating the application of our framework:

- **Warm-up Estimation Error (WEE)** measures the accuracy of the warm-up iterations estimated by the framework. Specifically, it represents the absolute difference, in seconds, between the actual steady-state (st) and the end of the warm-up phase as estimated by our framework. This metric helps us understand how closely the warm-up iterations provided by our framework align with the actual warm-up phase of the microbenchmark.
- **Measurement Deviation** evaluates the quality of the performance test results provided by the framework. For a given microbenchmark, this metric compares the set of performance measurements $\mathcal{M}$ returned by the framework to the ground-truth steady-state measurements $\mathcal{M}^*$. The set $\mathcal{M}^*$ consists of all measurements gathered from the steady-state iteration st onwards, for every time series of a particular microbenchmark[4]. A large deviation between $\mathcal{M}$ and $\mathcal{M}^*$ indicates poor result quality. To assess this deviation, we adhere to performance engineering best practices [29, 37, 56, 62] by calculating the confidence interval for the execution time ratio [32]. Specifically, we employ the bootstrap method [14, 31] with 10,000 iterations [24], using hierarchical random resampling with replacement at two levels [31], and a significance level of $\alpha = 0.05$. If the confidence interval for the ratio includes 1, there is no statistically significant difference between $\mathcal{M}$ and $\mathcal{M}^*$. Conversely, if the interval does not include 1, it suggests that $\mathcal{M}$ does not reflect the observed steady-state performance and could therefore mislead performance evaluation. For instance, a confidence interval of $(1.04, 1.06)$ indicates that $\mathcal{M}$ statistically differ from $\mathcal{M}^*$ by $5\% \pm 1\%$ with 95% confidence.
- **Testing Time** represents the total duration, in seconds, required to execute the microbenchmark using our framework. This duration includes both the warm-up iterations and the subsequent iterations used to collect performance measurements.

[4]Time series where the steady-state is not reached are excluded.

4.4 Implementation details

To facilitate the convergence of TSC models, we standardize each performance measurement x within each segment of the training dataset using the formula $(x - \mu)/\sigma$, where μ is the segment mean and σ is the segment standard deviation. We implement each of the TSC model as follows:

- **FCN**: We reimplement the neural network using TensorFlow[5]. Our implementation follows the hyperparameters specified in the original paper [59]. Specifically, we use three 1-D kernels with sizes {8, 5, 3} and three convolution blocks with filter sizes {128, 256, 128}.
- **OSCNN**: We reuse the original implementation provided in the replication package of Tang *et al.* [53], which uses PyTorch[6]. We maintained the default hyperparameter settings defined by the authors.
- **ROCKET**: We leverage the implementation provided by aeon[7]. We set the number of kernels to 500.

When training the neural networks (*i.e.,* FCN and OSCNN), we set aside 25% of the training set as a validation set. Both FCN and OSCNN are trained using the Adam optimizer [35] with a learning rate of 0.001, β_1=0.9, β_2=0.999, and ϵ=1e^{-8}. If the validation loss does not improve after 20 epochs, we halve the learning rate. We train the neural networks for up to 500 epochs, employing an early stopping strategy that halts training if the validation loss does not improve for 50 consecutive epochs. Throughout the training process, we save the model weights that yield the best validation loss. These optimized weights are then used for evaluation.

Since both FCN and OSCNN return probabilities rather than class labels, we tune the decision threshold to optimize Youden's index [61] on the validation set, rather than relying on a traditional 0.5 threshold. This tuning is not applicable to ROCKET, as it directly returns class labels.

The ROCKET training process does not use an explicit validation set, however, the implementation provided by aeon internally utilizes leave-one-out cross-validation for training the Ridge classifier.

For all the TSC models, we use a batch size of 1,024.

5 RESULTS

In this section, we discuss the study results by posing and answering three research questions.

5.1 RQ$_1$: To what extent can TSC models accurately classify stable and unstable measurements?

Motivation. We aim to evaluate the capabilities of TSC models in classifying measurements gathered from the steady-state and warm-up phases of microbenchmark execution. These results provide initial insights into the potential suitability of TSC models for dynamically halting warm-up iterations.

[5]https://www.tensorflow.org
[6]https://pytorch.org
[7]https://www.aeon-toolkit.org

Table 2: Results for the classification of segments (RQ_1).

Model	Prec.	Rec.	F1	Bal. Acc.
FCN	0.880	0.659	0.753	0.712
OSCNN	0.886	0.650	0.748	0.715
ROCKET	0.810	0.932	0.867	0.682

Approach. For each of the three TSC models, we compute the average precision, recall, F1-score and balanced accuracy across the five folds.

Results. Table 2 shows the results of the TSC models. Overall, we find that TSC models demonstrate good prediction accuracy in classifying *stable* and *unstable* segments, with F1-scores ranging from 0.748 to 0.867 and balanced accuracy between 0.682 and 0.712. ROCKET stands out as the best-performing model in terms of F1-score, while neural network models achieve higher balanced accuracy. The lower F1-scores of FCN and OSCNN are primarily due to their lower recall rates, which are 0.659 and 0.65, respectively. Conversely, ROCKET shows a notably high recall on stable segments (0.932) but has a balanced accuracy of only 0.682, which indicates a limited recall on *unstable* segments (~0.43). This behavior can be attributed to the higher number of false positives predicted by ROCKET, which is also reflected in its lower precision scores compared to FCN and OSCNN (0.81 *vs.* 0.88 and 0.886). It is important to note that false positives can be quite detrimental to our framework, as they may erroneously stop the microbenchmark execution, with no way to recover from the false prediction. In contrast, false negatives are less problematic since they do not halt execution, thus giving the framework another chance to correctly classify the stable measurements in the subsequent iteration. Nonetheless, in the following research questions, we will examine how the prediction accuracy of different models influences the overall effectiveness of our framework.

Answer to RQ_1: TSC models effectively classify *stable* and *unstable* measurements, so demonstrating their suitability for dynamically halting warm-up iterations. This supports their integration into our framework.

5.2 RQ_2: How does our AI-based framework compare to the state-of-practice (SOP) in Java microbenchmarking?

Motivation. Developers typically use a fixed number of warm-up iterations that are defined beforehand based on their domain expertise. With this research question, we aim to understand if the use of our framework for dynamically halting warm-up iterations provides advantages over the SOP.

Approach. We extract the number of warm-up iterations specified by the developers for each microbenchmark using a method similar to [26, 55]. We then compare the WEE of our framework with that of the SOP for each time series that reaches a steady-state. For this comparison, we employ the Wilcoxon signed-rank test [60] along with two measures of effect size: the Vargha-Delaney $\hat{A}_{12}$ [57] and the matched pairs rank biserial correlation r [33]. In our context, the $\hat{A}_{12}$ measures the proportion of pairs where the WEE of our framework is lower than that of the developers. An $\hat{A}_{12}$ value > 0.5 indicates that our framework provides a more accurate estimation of the warm-up phase than the SOP. The matched pairs rank biserial correlation r represents the difference between the proportion of favorable and unfavorable evidence; in our case, favorable evidence indicates a lower WEE for the framework. Thus, an $r > 0$ indicates that our framework performs better than the SOP.

Table 3: Results for the WEE comparison of models *vs.* SOP (RQ_2).

Model *vs.* SOP	p-value	$\hat{A}_{12}$	r
FCN *vs.* SOP	<0.001	0.664	0.352
OSCNN *vs.* SOP	<0.001	0.658	0.348
ROCKET *vs.* SOP	<0.001	0.683	0.282

In addition to evaluating the accuracy of the warm-up estimation, we examine how the adoption of the framework affects the quality of performance test results and the testing time for each microbenchmark. To evaluate the impact of our framework on result quality, we calculate the percentage of microbenchmarks that show an improvement (or regression) compared to the SOP. For a given microbenchmark, an improvement in result quality occurs under the following two conditions: (i) the performance measurements from SOP ($\mathcal{M}_{SOP}$) are statistically different from those of the steady-state ($\mathcal{M}^*$) (*i.e.,* the confidence interval for the execution time ratio does not include 1), and (ii) the performance measurements provided by the framework $\mathcal{M}$ are not statistically different from $\mathcal{M}^*$ (*i.e.,* the confidence interval includes 1). This indicates that the framework improves result quality, by providing measurements that more faithfully represent the microbenchmark true steady-state performance. A result quality regression indicates the opposite situation.

We also report the percentage of microbenchmarks where the framework shows improvement (or regression) in terms of testing time. A microbenchmark is considered improved if the framework reduces its testing time when compared to SOP. Note that we only consider testing time improvements (or regressions) in cases where the measurements provided by both the framework and the SOP are not statistically different from the steady-state. This ensures that a reduction in testing time is not mistakenly interpreted as an improvement if it leads to poor result quality.

To achieve a fair comparison, we derive the set of performance measurements $\mathcal{M}$ returned by the framework, as well as the microbenchmark testing time, using the same number of measurement iterations and JMH forks specified by the developers (more details on this process are available in our replication package).

Results. We discuss the results of this RQ from different aspects.

Warm-up estimation accuracy. Table 3 shows the results of the comparison between the WEE of the framework and the SOP. We observe that the framework delivers more accurate estimates of the warm-up phase across all employed TSC models. The results of the Wilcoxon test show that the differences are statistically significant ($p < 0.001$) for all models, with medium effect sizes ($\hat{A}_{12}$ ranging from 0.658 to 0.683). Additionally, the matched pairs rank biserial

Table 4: Percentages of improvement and regression when comparing models *vs.* SOP (RQ_2).

Model *vs.* SOP	Improvement (%)			Regression (%)			Net Improvement (%)
	Res. Quality	Testing Time	***Total***	Res. Quality	Testing Time	***Total***	(***Tot. Impr. - Tot. Regr.***)
FCN *vs.* SOP	16.7	30.7	***47.4***	14.3	7.8	***22.2***	***+25.3***
OSCNN *vs.* SOP	17.9	30.9	***48.8***	13.7	8.2	***21.8***	***+27.0***
Rocket *vs.* SOP	9.4	20.1	***29.5***	25.9	6.5	***32.4***	***-2.9***

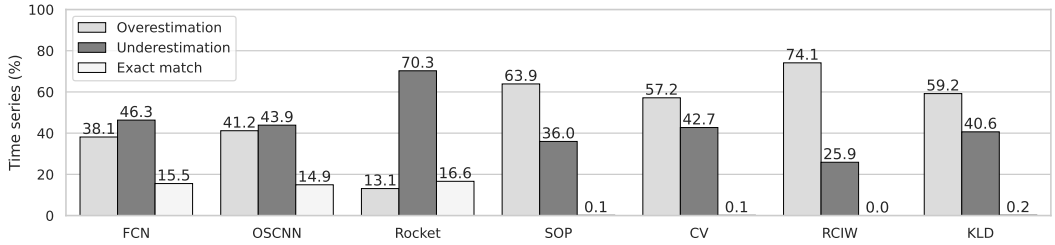

Figure 4: Percentages of *overestimation*, *underestimation*, and *exact match* for models/SOP/SOTA ($RQ_{2/3}$).

correlation results emphasize a considerable gap between the favorable and unfavorable outcomes, with positive r values ranging from 0.282 to 0.352. This indicates a higher likelihood of achieving lower WEE with our framework.

Result quality. Table 4 presents the percentages of improvements and regressions in result quality and testing time achieved by our framework compared to the SOP. We observe that our framework enhances result quality in 16.7%, 17.9%, and 9.4% of the microbenchmarks when using FCN, OSCNN, and ROCKET, respectively. However, it also exhibits a number of regressions, ranging from 13.7% to 25.9%.

ROCKET yields the worst result quality among the TSC models, with only 9.4% improvements and 25.9% regressions. This poor result quality is likely due to its tendency to underestimate the warm-up phase, thus leading to the collection of measurements that significantly deviate from the true steady-state performance. As illustrated in Fig. 4, ROCKET underestimates the warm-up phase in 70.3% of the time series, while the SOP only does so in 36% of cases. This propensity to stop the microbenchmark earlier could be linked to the higher number of false positives produced by ROCKET (see RQ_1 discussion).

In contrast, OSCNN provides the best result quality among the TSC models, showing a slight tendency towards improvement compared to SOP, with 17.9% improvements and 13.7% regressions. Additionally, the degree of measurement deviation is lower in OSCNN, as evident from Table 5. This table presents descriptive statistics of the relative measurement deviation, which we use to quantify the magnitude of deviation from the steady-state. This metric is calculated as the absolute value of the difference between the center of the confidence interval for the execution time ratio and one. The table shows that the measurements returned by OSCNN deviate less from the steady-state than those of SOP, with a median deviation of 5.5% (*vs.* 6.5% in SOP) and an interquartile range (IQR) of 2.4-12.3% (*vs.* 2.4-19.2%).

Testing time. From Table 4, we observe that our framework reduces the testing time of SOP across a large percentage of microbenchmarks, showing improvements by 30.7%, 30.9%, and 20.1% for FCN, OSCNN, and ROCKET, respectively. By looking at Table 5, we also note that using the framework significantly reduces the median testing time of the SOP by 21% to 49%. For instance, FCN shows a median (IQR) testing time of 74 (43-150) seconds, while the SOP provides a median testing time of 100 (30-403) seconds. These findings indicate that, for a significant portion of microbenchmarks, our framework drastically reduces the testing time of the SOP without compromising the result quality. This enhancement can be attributed to the higher accuracy achieved by the framework in estimating the end of the warm-up phase. Indeed, as shown in Fig. 4, SOP overestimates the end of the warm-up phase in 63.9% of cases, thereby increasing the testing time. In contrast, our framework overestimates it in only 38.1% (FCN), 41.2% (OSCNN), and 13% (ROCKET) of the cases, respectively. Moreover, we observe that our framework can exactly identify the end of the warm-up phase in 15.5%, 14.9%, and 16.6% of the cases, respectively, whereas the SOP does so in only 0.1% of the cases.

Overall. We find that the framework improves either the result quality or the testing time of the SOP in 47.4%, 48.8%, and 29.5% of the microbenchmarks, depending on the TSC model employed (see Table 4). Conversely, it shows regressions in 22.2%, 21.8%, and 32.4% of the cases. This yields a net improvement of +25.3% for FCN and +27% for OSCNN, and a net regression of -2.9% for ROCKET.

Answer to RQ_2: Our framework provides more accurate estimates of the warm-up phase compared to the SOP. This higher accuracy translates into a net improvement in either result quality or testing time in up to +27% of the microbenchmarks, with OSCNN demonstrating the highest net improvement.

Table 5: Relative measurement deviation and testing time of models, SOP, and SOTA ($RQ_{2/3}$).

Model/SOP/SOTA	Rel. Meas. Dev. (%) Median (IQR)	Testing Time (*sec*) Median (IQR)
FCN	5.7 (2.8-11.3)	74 (43-150)
OSCNN	5.5 (2.4-12.3)	79 (47-161)
Rocket	10.0 (4.5-21.7)	51 (30-78)
SOP	6.5 (2.4-19.2)	100 (30-403)
CV	9.9 (4.7-17.5)	75 (51-113)
RCIW	4.5 (1.8-11.5)	300 (265-301)
KLD	7.9 (4.2-13.4)	121 (94-156)

5.3 RQ_3: How does our AI-based framework compare to the state-of-the-art (SOTA) in Java microbenchmarking?

Motivation. This research question aims to compare our framework against prior SOTA approaches that dynamically halt warm-up iterations. Our objective is to assess whether the use of sophisticated TSC models provides benefits over traditional dynamic approaches.

Approach. We compare the WEE of our framework to the dynamic technique proposed by Laaber *et al.* [37], by evaluating all three variants of their technique: COV, RCIW, and KLD. To determine the warm-up iterations for each variant, we use the original replication package provided by the authors [37]. As in RQ_2, we employ the Wilcoxon signed-rank test [60], the Vargha-Delaney $\hat{A}_{12}$, and the matched pairs rank biserial correlation r to assess whether our framework provides more accurate estimates of the warm-up phase.

Additionally, we report the percentages of improvements and regressions in both results quality and testing time. For an unbiased comparison, we derive the set of performance measurements, $\mathcal{M}$, and the testing time for our framework using the same measurement window defined by Laaber *et al.* (*i.e.,* 100 measurement iterations in our setup) and the same number of JMH forks (the number of forks is dynamically estimated for each microbenchmark in Laaber *et al.*'s technique).

Results. Similarly to RQ2, we discuss the results of this RQ from various perspectives.

Warm-up estimation accuracy. Table 6 shows the results of the comparison between the WEE provided by the framework and those of SOTA. We find that the framework provides more accurate estimations of the warm-up phase across all the models than all SOTA variants ones. The differences are statistically significant ($p < 0.001$) for all comparisons, based on the results of the Wilcoxon test. The $\hat{A}_{12}$ values range from 0.621 to 0.731, indicating a small to large effect size according to the thresholds defined by Vargha and Delaney [57]. The rank biserial correlation r ranges from 0.157 to 0.414, suggesting a prevalence for the favorable outcome.

Result quality. Table 7 presents the percentages of improvement and regression in result quality and testing time. We observe that ROCKET performs worse than SOTA approaches in terms of result quality. Specifically, ROCKET produces regressions in 24.6%, 51.4%, and 22.5% of the microbenchmarks when compared to CV, RCIW, and KLD, respectively. Moreover, it reports improvements in only 11.3%, 4.1%, and 7.2% of the cases. This limitation might once again be attributed to the higher false positive rate produced by this model. In contrast, we observe that neural network models generally perform better than SOTA in terms of result quality, with the only exception being RCIW. For instance, OSCNN improves the result quality of CV and KLD in 28.8% and 27.8% of microbenchmarks, respectively, and causes regressions in 12.3% and 12.1% of them. Moreover, OSCNN produces lower relative measurement deviations than CV and KLD, as shown in Table 5. The median (IQR) deviation for OSCNN is 5.5% (2.4-12.3%), while CV and KLD induce deviations of 9.9% (4.7-17.5%) and 7.9% (4.2-13.4%), respectively. We observe a similar trend of improvements in FCN, albeit with slightly less pronounced results.

Table 6: Results for the WEE comparison of models *vs.* SOTA (RQ_3).

Model *vs.* SOTA	p-value	$\hat{A}_{12}$	r
FCN *vs.* CV	<0.001	0.637	0.370
FCN *vs.* RCIW	<0.001	0.731	0.400
FCN *vs.* KLD	<0.001	0.630	0.292
OSCNN *vs.* CV	<0.001	0.632	0.385
OSCNN *vs.* RCIW	<0.001	0.728	0.414
OSCNN *vs.* KLD	<0.001	0.621	0.288
ROCKET *vs.* CV	<0.001	0.649	0.245
ROCKET *vs.* RCIW	<0.001	0.726	0.264
ROCKET *vs.* KLD	<0.001	0.656	0.157

The framework provides lower result quality than RCIW across all TSC models, causing regressions in 24.2% (FCN), 21.8% (OSCNN), and 51.4% (ROCKET) of the microbenchmarks. This behavior can be attributed to the RCIW's higher tendency to overestimate the warm-up phase, observed in 74.1% of the cases (see Fig. 4), which increases the chance of gathering steady-state measurements. While this tendency ensures better result quality compared to our framework, it also leads to longer testing times.

Testing time. As shown in Table 5, RCIW reports a median (IQR) testing time of 300 (265-301) seconds, while the most time-consuming TSC model of our framework, OSCNN, produces a median (IQR) testing time of 79 (47-161) seconds, which is approximately one-quarter of RCIW testing time. Furthermore, from Table 7, we observe that FCN and OSCNN can improve the testing time for about half of the microbenchmarks (50.3% and 52.6%, respectively) without affecting result quality. When compared to the other two SOTA variants, CV and KLD, our framework still shows improvements in testing time. For example, OSCNN reduces the testing time in 26.8% and 23% of the microbenchmarks, respectively.

Overall. We find that, when employing neural network models such as FCN and OSCNN, our framework provides improvements over the SOTA in either result quality or testing time in approximately half of the microbenchmarks, with percentages ranging from 50.3% to 59.6%. The percentages of regressions are lower, with values ranging from 20% to 28.8%, thus resulting in substantial net improvements. For instance, from Table 7, we can observe that

Table 7: Percentages of improvement and regression when comparing models *vs.* SOTA (RQ_3).

Model *vs.* SOTA	Improvement (%)			Regression (%)			Net Improvement (%)
	Res. Quality	Testing Time	***Total***	Res. Quality	Testing Time	***Total***	(***Tot. Impr. - Tot. Regr.***)
FCN *vs.* CV	26.8	27.1	***53.9***	12.3	7.7	***20.0***	***+34.0***
FCN *vs.* RCIW	6.3	50.3	***56.7***	24.2	4.6	***28.8***	***+27.8***
FCN *vs.* KLD	25.9	24.4	***50.3***	13.1	10.4	***23.5***	***+26.8***
OSCNN *vs.* CV	28.8	26.8	***55.6***	12.3	8.0	***20.3***	***+35.3***
OSCNN *vs.* RCIW	7.0	52.6	***59.6***	21.8	4.8	***26.6***	***+32.9***
OSCNN *vs.* KLD	27.8	23.0	***50.9***	12.1	12.8	***24.9***	***+25.9***
ROCKET *vs.* CV	11.3	19.8	***31.1***	24.6	2.7	***27.3***	***+3.8***
ROCKET *vs.* RCIW	4.1	24.4	***28.5***	51.4	3.4	***54.8***	***-26.3***
ROCKET *vs.* KLD	7.2	21.5	***28.7***	22.5	3.9	***26.5***	***+2.2***

OSCNN provides a net improvement over CV, RCIW, and KLD by +35.3%, +32.9%, and +25.9%, respectively.

Answer to RQ_3: Our framework provides more accurate estimates of the warm-up phase than the SOTA. While ROCKET often reduces result quality, variants of the framework based on neural network models observably enhance either the result quality or testing time of the SOTA techniques, leading to net improvements in up to +35.3% of the microbenchmarks.

6 THREATS TO VALIDITY

Construct validity. We derive the number of warm-up iterations through post-hoc analysis using a methodology similar to [37, 55]. This evaluation methodology does not account for the overhead introduced by the TSC model prediction. However, we do not consider this overhead for both our framework and the SOTA techniques when calculating the testing time, whereas SOP does not involve any overhead, thus ensuring a fair comparison. Moreover, to further mitigate this threat, we measured the inference time of each TSC model on a sample of 500 measurement segments, and we obtained a median overhead per microbenchmark iteration of 9% for FCN, 2% for OSCNN, and 2% for ROCKET. These overheads appear minimal when compared to the testing time differences observed in Table 5. Therefore, they are unlikely to impact our main findings.

We rely on the notion of steady-state as defined by Barrett *et al.* [6], whereas using an alternative notion may change the study outcomes. We integrate three state-of-the-art TSC models into our framework, using different models may yield different results.

The proposed framework aims to dynamically infer the appropriate number of warm-up iterations at runtime. Although other relevant parameters, such as the number of forks and measurement iterations, can influence the quality of results and the testing time of microbenchmarks, these aspects are beyond the scope of our work. Nonetheless, to ensure fairness in our evaluation, we consistently use the same number of forks and measurement iterations when comparing our approach with baselines.

Internal validity. TSC model training involves randomness, hence there might be slight differences in the results when re-executing the experiments. We employ TSC models by using default hyper-parameters, whereas using different hyper-parameters may change the experiments outcomes.

External validity. The findings of our study may not generalize to microbenchmarks beyond our experiment dataset. However, we evaluate our framework on 583 microbenchmarks from 30 well-established OSS projects spanning various domains (see Table 1). Furthermore, the number of systems involved in our study is larger than the ones considered in most recent software performance studies [7, 16, 29, 37, 38].

Each TSC model is trained on an average of 417,520 segments from 468 microbenchmarks per fold iteration. Using training sets of different sizes and heterogeneity could affect the prediction accuracy of the models, and consequently, the outcome of the framework. We encourage future dedicated analyses to investigate how these factors might affect the framework effectiveness.

We utilize the dataset of JMH measurements from our previous work [55], which was collected using a rigorous procedure to minimize noise. We specifically choose these measurements to reduce confounding factors that could compromise the soundness of our study. However, using measurements from more variable environments (*e.g.,* cloud) could lead to different study outcomes.

7 RELATED WORK

Besides the works discussed in Section 2, other techniques have been proposed to estimate the attainment of steady-state in performance tests. Kalibera and Jones [32] introduced a methodology that relies on the visual analysis of auto-correlation function plots, lag plots, and run-sequence plots. However, this technique requires manual analysis, making it impractical for real-world use cases. Barrett *et al.* [6] proposed an automated technique based on change point analysis [34] to identify shifts in the time series of execution times and determine the attainment of the steady-state. While this technique is highly useful in scientific settings where a rigorous notion of steady-state is necessary, it is also impractical for real-world use due to the significant time effort required to run the performance tests. AlGhamdi *et al.* [2] proposed a technique to stop load tests when performance metric values become repetitive. He *et al.* [23] proposed a statistical approach to halt performance tests in the cloud. Abdullah *et al.* [1] proposed an approach for the early stopping of non-productive experiments in performance testing.

8 CONCLUSION

In this paper, we propose and study an AI-based framework to dynamically stop warm-up iterations in Java performance tests. The results show that, when integrated with certain TSC models, our framework delivers result quality comparable to the state-of-practice while drastically reducing the testing time. Furthermore, we find that the framework improves both the result quality and testing time in most of the investigated state-of-the-art techniques. This higher effectiveness can be attributed to the ability of TSC models to capture complex, non-linear patterns in measurements associated with steady-state execution, which simple statistical measures—such as those employed by state-of-the-art techniques–may overlook.

Our work has significant implications for both practitioners and researchers. Practitioners can use our framework to ensure result quality, while significantly reducing testing time of performance testing suites. Researchers can integrate our framework into advanced software engineering techniques (such as, genetic improvement [39, 49], configuration tuning [9], or software self-adaptation [58]), where rigorous and time-efficient performance evaluation is crucial. Additionally, researchers can leverage our framework to experiment with different (potentially more effective) TSC models, or use it as a robust baseline for future techniques that dynamically stop warm-up iterations at runtime. Despite the promising results, our findings still highlight room for improvement in this area, particularly in the enhancement of the quality of performance test results. We encourage future research efforts aimed at this goal.

ACKNOWLEDGMENTS

This work was supported by the Italian Government (Ministero dell'Università e della Ricerca, PRIN 2022 PNRR) under the project "RECHARGE: Monitoring, Testing, and Characterization of Performance Regressions" (cod. P2022SELA7). Additional support was provided by the "ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing", funded by the European Union – NextGenerationEU. This work also received support from the research start-up project "Enhancing Software Performance Testing using Artificial Intelligence", funded by the University of L'Aquila.

REFERENCES

[1] Milad Abdullah, Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, and Petr Tůma. 2023. Early Stopping of Non-productive Performance Testing Experiments Using Measurement Mutations. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 86–93. https://doi.org/10.1109/SEAA60479.2023.00022

[2] Hammam M. AlGhamdi, Cor-Paul Bezemer, Weiyi Shang, Ahmed E. Hassan, and Parminder Flora. 2023. Towards reducing the time needed for load testing. *Journal of Software: Evolution and Process* 35, 3 (2023), e2276. https://doi.org/10.1002/smr.2276 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2276 e2276 smr.2276.

[3] Hammam M. Alghmadi, Mark D. Syer, Weiyi Shang, and Ahmed E. Hassan. 2016. An Automated Approach for Recommending When to Stop Performance Tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 279–289. https://doi.org/10.1109/ICSME.2016.46

[4] Monica Arul and Ahsan Kareem. 2021. Applications of shapelet transform to time series classification of earthquake, wind and wave data. *Engineering Structures* 228 (2021), 111564. https://doi.org/10.1016/j.engstruct.2020.111564

[5] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* 31, 3 (2017), 606–660. https://doi.org/10.1007/s10618-016-0483-9

[6] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

[7] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2022. PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1529–1544. https://doi.org/10.1109/TSE.2020.3023955

[8] Kaixuan Chen, Dalin Zhang, Lina Yao, Bin Guo, Zhiwen Yu, and Yunhao Liu. 2021. Deep Learning for Sensor-based Human Activity Recognition: Overview, Challenges, and Opportunities. *ACM Comput. Surv.* 54, 4, Article 77 (may 2021), 40 pages. https://doi.org/10.1145/3447744

[9] Tao Chen and Miqing Li. 2021. Multi-objectivizing software configuration tuning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 453–465. https://doi.org/10.1145/3468264.3468555

[10] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2017. Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 243–252. https://doi.org/10.1109/ICSE-SEIP.2017.26

[11] Zhicheng Cui, Wenlin Chen, and Yixin Chen. 2016. Multi-Scale Convolutional Neural Networks for Time Series Classification. *CoRR* abs/1603.06995 (2016). arXiv:1603.06995 http://arxiv.org/abs/1603.06995

[12] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/2451116.2451141

[13] Hoang Anh Dau, Anthony J. Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, and Eamonn J. Keogh. 2018. The UCR Time Series Archive. *CoRR* abs/1810.07758 (2018). arXiv:1810.07758 http://arxiv.org/abs/1810.07758

[14] Anthony Christopher Davison and David Victor Hinkley. 1997. *Bootstrap methods and their application*. Cambridge University Press.

[15] Angus Dempster, François Petitjean, and Geoffrey I. Webb. 2020. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery* 34, 5 (2020), 1454–1495. https://doi.org/10.1007/s10618-020-00701-z

[16] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1435–1446. https://doi.org/10.1145/3377811.3380351

[17] B.S. Everitt and A. Skrondal. 2010. *The Cambridge Dictionary of Statistics*. Cambridge University Press.

[18] Mikael Fagerström, Emre Emir Ismail, Grischa Liebel, Rohit Guliani, Fredrik Larsson, Karin Nordling, Eric Knauss, and Patrizio Pelliccione. 2016. Verdict Machinery: On the Need to Automatically Make Sense of Test Results. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 225–234. https://doi.org/10.1145/2931037.2931064

[19] Navid Mohammadi Foumani, Lynn Miller, Chang Wei Tan, Geoffrey I. Webb, Germain Forestier, and Mahsa Salehi. 2024. Deep Learning for Time Series Classification and Extrinsic Regression: A Current Survey. *ACM Comput. Surv.* (feb 2024). https://doi.org/10.1145/3649448 Just Accepted.

[20] Kunihiko Fukushima. 1969. Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements. *IEEE Transactions on Systems Science and Cybernetics* 5, 4 (1969), 322–333. https://doi.org/10.1109/TSSC.1969.300225

[21] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36, 4 (1980), 193–202. https://doi.org/10.1007/BF00344251

[22] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[23] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 188–199. https://doi.org/10.1145/3338906.3338912

[24] Tim C. Hesterberg. 2015. What Teachers Should Know About the Bootstrap: Resampling in the Undergraduate Statistics Curriculum. *The American Statistician* 69, 4 (2015), 371–386. https://doi.org/10.1080/00031305.2015.1089789

arXiv:https://doi.org/10.1080/00031305.2015.1089789 PMID: 27019512.

[25] Donald E. Hilt and Donald W. Seegrist. 1977. *Ridge, a computer program for calculating ridge regression estimates*. Vol. no.236. Upper Darby, Pa, Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977. 10 pages. https://www.biodiversitylibrary.org/item/137258 https://www.biodiversitylibrary.org/bibliography/68934.

[26] Muhammad Imran, Vittorio Cortellessa, Davide Di Ruscio, Riccardo Rubei, and Luca Traini. 2024. An Empirical Study on Code Coverage of Performance Testing. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 48–57. https://doi.org/10.1145/3661167.3661196

[27] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 448–456.

[28] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery* 33, 4 (2019), 917–963. https://doi.org/10.1007/s10618-019-00619-1

[29] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. 2023. Automated Generation and Evaluation of JMH Microbenchmark Suites From Unit Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1704–1725. https://doi.org/10.1109/TSE.2022.3188005

[30] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[31] Tomas Kalibera and Richard Jones. 2012. *Quantifying Performance Changes with Effect Size Confidence Intervals*. Technical Report 4–12. University of Kent. 55 pages. http://www.cs.kent.ac.uk/pubs/2012/3233

[32] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management* (Seattle, Washington, USA) *(ISMM '13)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/2464157.2464160

[33] Dave S. Kerby. 2014. The Simple Difference Formula: An Approach to Teaching Nonparametric Correlation. *Comprehensive Psychology* 3 (2014), 11.IT.3.1. https://doi.org/10.2466/11.IT.3.1 arXiv:https://doi.org/10.2466/11.IT.3.1

[34] R. Killick, P. Fearnhead, and I. A. Eckley. 2012. Optimal Detection of Changepoints With a Linear Computational Cost. *J. Amer. Statist. Assoc.* 107, 500 (2012), 1590–1598. http://www.jstor.org/stable/23427357

[35] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[36] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79 – 86. https://doi.org/10.1214/aoms/1177729694

[37] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time without Sacrificing Result Quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 989–1001. https://doi.org/10.1145/3368089.3409683

[38] Christoph Laaber, Tao Yue, and Shaukat Ali. 2024. Evaluating Search-Based Software Microbenchmark Prioritization. *IEEE Transactions on Software Engineering* (2024), 1–16. https://doi.org/10.1109/TSE.2024.3380836

[39] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2015), 118–135. https://doi.org/10.1109/TEVC.2013.2281544

[40] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/3030207.3030213

[41] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network In Network. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.4400

[42] Jason Lines and Anthony Bagnall. 2015. Time series classification with ensembles of elastic distance measures. *Data Mining and Knowledge Discovery* 29, 3 (01 May 2015), 565–592. https://doi.org/10.1007/s10618-014-0361-2

[43] Sourav Majumdar and Arnab Kumar Laha. 2020. Clustering and classification of time series using topological data analysis with applications to finance. *Expert Systems with Applications* 162 (2020), 113868. https://doi.org/10.1016/j.eswa.2020.113868

[44] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

[45] Matthew Middlehurst, Patrick Schäfer, and Anthony J. Bagnall. 2023. Bake off redux: a review and experimental evaluation of recent time series classification algorithms. *CoRR* abs/2304.13029 (2023). https://doi.org/10.48550/ARXIV.2304.13029 arXiv:2304.13029

[46] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[47] Scott Oaks. 2014. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* " O'Reilly Media, Inc.".

[48] OpenJDK. 2014. Java Microbenchmarking Harness (JMH). https://github.com/openjdk/jmh/. Accessed: 2024-09-09.

[49] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[50] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew Dai, Nissan Hajaj, Peter Liu, Xiaobing Liu, Mimi Sun, Patrik Sundberg, Hector Yee, Kun Zhang, Gavin Duggan, Gerardo Flores, Michaela Hardt, Jamie Irvine, Quoc Le, Kurt Litsch, Jake Marcus, Alexander Mossin, and Jeff Dean. 2018. Scalable and accurate deep learning for electronic health records. *npj Digital Medicine* 1 (01 2018). https://doi.org/10.1038/s41746-018-0029-1

[51] Patrick Schäfer. 2015. The BOSS is concerned with time series classification in the presence of noise. *Data Mining and Knowledge Discovery* 29, 6 (2015), 1505–1530. https://doi.org/10.1007/s10618-014-0377-7

[52] Chang Wei Tan, Angus Dempster, Christoph Bergmeir, and Geoffrey I. Webb. 2022. MultiRocket: multiple pooling operators and transformations for fast and effective time series classification. *Data Min. Knowl. Discov.* 36, 5 (sep 2022), 1623–1646. https://doi.org/10.1007/s10618-022-00844-1

[53] Wensi Tang, Guodong Long, Lu Liu, Tianyi Zhou, Michael Blumenstein, and Jing Jiang. 2022. Omni-Scale CNNs: a simple and effective kernel size configuration for time series classification. In *International Conference on Learning Representations*. https://openreview.net/forum?id=PDYs7Z2XFGv

[54] Luca Traini. 2022. Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study. *Empirical Software Engineering* 27, 3 (2022), 74. https://doi.org/10.1007/s10664-021-10069-3

[55] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2022), 13. https://doi.org/10.1007/s10664-022-10247-x

[56] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 25 (dec 2021), 23 pages. https://doi.org/10.1145/3485136

[57] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[58] Shu Wang, Henry Hoffmann, and Shan Lu. 2022. AgileCtrl: a self-adaptive framework for configuration tuning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 459–471. https://doi.org/10.1145/3540250.3549136

[59] Zhiguang Wang, Weizhong Yan, and Tim Oates. 2017. Time series classification from scratch with deep neural networks: A strong baseline. In *2017 International Joint Conference on Neural Networks (IJCNN)*. 1578–1585. https://doi.org/10.1109/IJCNN.2017.7966039

[60] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[61] W. J. Youden. 1950. Index for rating diagnostic tests. *Cancer* 3, 1 (1950), 32–35. https://doi.org/"https://doi.org/10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3"

[62] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1495–1507. https://doi.org/10.1109/ICSE48619.2023.00130