AMBER: AI-Enabled Java Microbenchmark Harness

Antonio Trovato
University of Salerno
Salerno, Italy
atrovato@unisa.it

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

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

Dario Di Nucci
University of Salerno
Salerno, Italy
ddinucci@unisa.it

***Abstract*—JMH is the standard framework for developing and running Java microbenchmarks—lightweight performance tests used to evaluate the execution time of small Java code segments. A key challenge in designing JMH microbenchmarks is determining the appropriate number of warm-up iterations–repeated executions needed to bring microbenchmarks to a performance steady state. Too few warm-up iterations can compromise result quality, as performance measurements may not accurately reflect steady-state behavior. Conversely, too many warm-up iterations can unnecessarily increase testing time. Here, we present AMBER, an AI-enabled extension of JMH, which leverages Time Series Classification algorithms to predict the beginning of the steady-state phase at run-time and dynamically halt warm-up iterations accordingly. Empirical results show the potential of AMBER in enhancing the cost-effectiveness of Java microbenchmarks. A demo video of AMBER is available at https://www.youtube.com/watch?v=7zOngDQ1z_k.**

***Index Terms*—Performance Testing, Microbenchmark, JMH**

I. Introduction

Microbenchmarking is a type of small-scale performance testing widely used to evaluate the execution time of Java software [1]. This testing methodology involves repeatedly executing a small code segment while measuring its execution time. During the initial phase of execution, Java microbenchmarks are subject to a wide range of JVM optimizations, which make execution times highly unstable and potentially misleading [2]–[4]. Once these optimizations are completed, the microbenchmark reaches a steady-state performance [2], [5]. Therefore, a crucial factor in gathering quality measurements during microbenchmarking is properly configuring the number of "warm-up iterations" [3]–[5]–repeated executions performed to bring the microbenchmark to steady-state performance before starting to collect measurements.

Developers mostly rely on *Java Microbenchmark Harness* [6] (JMH), the de-facto standard for building and running Java microbenchmarks [1], [7], to "manually" configure the desired number of warm-up iterations (*state-of-practice*). These iterations are usually estimated based on the developers' domain expertise; however, based on recent studies [5], they frequently turn out to be incorrect. In line with the recent surge of research interest in JMH microbenchmarks [8]–[14], Laaber *et al.* [15] have proposed techniques for dynamically stopping warm-up iterations by applying thresholds to statistical properties of measurements (*e.g.,* coefficient of variation). These *state-of-the-art* techniques have demonstrated higher effectiveness than developers' manual configurations in identifying the actual end of the warm-up phase. However, the resulting warm-up iterations proved often suboptimal, leading to either reduced result quality or extended testing times [5].

In this paper, we present AMBER (**A**I-enabled Java **M**icro**be**nchmark Ha**r**ness), a tool that implements our AI-driven approach [16] to halt warm-up iterations at run-time dynamically. AMBER relies on Time Series Classification (TSC) algorithms to predict whether a set of measurements is stable or not, and it exploits this capability to stop warm-up iterations at run-time dynamically. We conducted an empirical evaluation of AMBER [16] on 586 microbenchmarks from 30 established Java Open-Source software while integrating three different state-of-the-art TSC algorithms, namely Fully Convolutional Network [17] (FCN), Omni-Scale Convolutional Network [18] (OSCNN), and Random Convolutional Kernel Transform [19] (ROCKET). Results show that AMBER noticeably improves the cost-effectiveness of performance testing. Specifically, when compared to both the state-of-practice and the state-of-the-art, it achieves a net improvement–in either result quality or testing time–in up to +27% and +35.3% of the microbenchmarks, respectively. AMBER extends the JMH framework, enabling practitioners and researchers to seamlessly adopt our AI-driven approach for dynamically halting warm-up iterations.

II. The Amber Approach

Fig. 1 provides a simplified visualization of the AMBER process, which comprises three main phases: (i) *Data Preprocessing*, (ii) *Model Training*, and (iii) *Application*. The initial phases are executed offline, while the final phase is performed during performance testing. During the *Data Preprocessing* phase, AMBER preprocesses a time series of performance measurements with a (known) predefined warm-up end. This step involves segmenting the time series into sub-intervals and labeling each as either *stable* or *unstable*, based on the presence of warm-up measurements. The labeled segments are subsequently used to train a Time Series Classification model (*Model Training*). Finally, the trained model is applied to dynamically halt warm-up iterations at runtime (*Application*).

This work has been partially funded by the European Union under NextGenerationEU with the *RECHARGE* research project, which has been funded by MUR PRIN 2022 PNRR program (Code: P2022SELA7). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or The European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

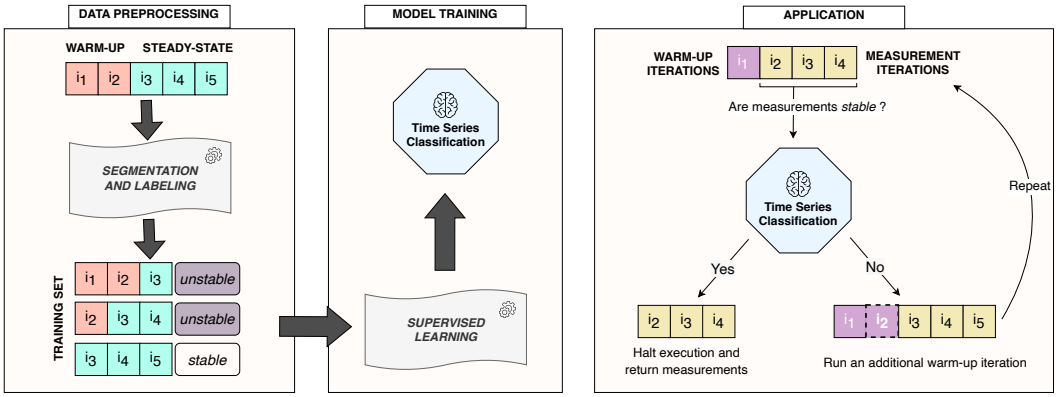


Fig. 1: Overview of the main phases of AMBER: *Data Preprocessing*, *Model Training* and *Application*.

A. Data Preprocessing

This phase focuses on preparing the training set for supervised learning. To accomplish this, AMBER begins with a time series of performance measurements and an annotation that indicates the end of the warm-up phase. The time series represents the observed execution time over sequential iterations of a JMH microbenchmark. The annotation identifies the iteration at which the performance steady state is achieved, following the methodology proposed by Barrett *et al.* [2].

To build our training set, we extract smaller, overlapping segments of fixed size from the original time series, each representing a contiguous block of performance measurements. These segments are then binary labeled as (i) *stable* if the segment includes only measurements taken during the steady state or (ii) *unstable* if the segment contains at least one measurement from the warm-up phase.

B. Model training

Our learning objective is to perform binary classification on segments of performance measurements. We employ TSC algorithms to categorize each segment as either *stable* or *unstable*. In particular, we explore three distinct TSC algorithms described in detail below.

FCN was introduced by Wang *et al.* [17] for classifying univariate time series. This neural network architecture functions as a feature extractor by stacking three convolutional layers, each accompanied by a batch normalization layer and a ReLU activation layer. The extracted features are processed through a global average pooling layer and fed into a softmax classifier to produce the final output label.

OSCNN introduces the Omni-Scale block [18], which employs a universal rule to automatically determine the kernel sizes for one-dimensional convolutional neural networks (1D-CNNs). This method facilitates selecting the optimal receptive field size, a crucial factor that significantly impacts the performance of 1D-CNNs in time series classification (TSC) [20].

ROCKET is a pipeline-based classifier [19] that utilizes a large set of randomly parameterized convolutional kernels to transform the input data. The transformation is achieved through two pooling operations: extracting the maximum value and calculating the proportion of positive values. These pooled features are combined into a feature vector for each kernel. The resulting feature vectors are then used to train a Ridge Classifier [21] through cross-validation.

C. Application

AMBER employs TSC models to dynamically identify when a microbenchmark reaches steady-state performance. During the execution, AMBER continuously monitors and analyzes incoming performance measurements using the TSC model. When the model detects an achieved steady state, AMBER immediately halts the microbenchmark execution and returns the current performance measurements for evaluation.

The core idea of our approach is leveraging TSC models to effectively distinguish between *stable* and *unstable* measurements so that AMBER can automatically terminate the microbenchmark execution when the warm-up phase concludes, ensuring high-quality results with minimal testing time.

III. AMBER ARCHITECTURE AND INNER WORKING

AMBER is implemented as an extension of JMH, introducing the ability to halt warm-up iterations during execution dynamically, which is achieved through the integration of a TSC algorithm, as described in section II-C.

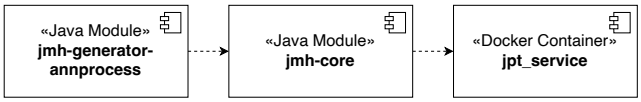


Fig. 2: Overview of the architecture of AMBER

Fig. 2 shows the architecture of AMBER. The figure shows the two core modules of JMH, namely `jmh-core` and `jmh-generator-annprocess`, as well as our extension that integrates the ability to dynamically halt warm-up iterations at run-time, namely `jpt-service`. The `jmh-generator-annprocess` generates the boilerplate code for Java microbenchmarks. Specifically, during compilation, it scans the methods and classes that are annotated

with JMH-specific annotations (like `@Benchmark`), and it generates microbenchmark code following JMH guidelines for reliable Java performance testing. `jmh-core` includes the core implementation of the microbenchmarking engine, which is responsible for executing benchmark methods, managing warm-up and measurement iterations, and collecting performance measurements. In addition, it defines JMH-specific annotations that users can apply to their code to configure aspects of their microbenchmarks, including the number of warm-up iterations (`@Warmup`). AMBER modifies the `jmh-core` module by introducing a `@DynamicHalt` annotation to enable dynamic halting of warm-up iterations. This annotation can be applied at either the class or method levels. When applied at the class level, the dynamic halting policy is enforced on all benchmark methods within the Java class. Conversely, when applied at the method level, the policy is restricted to the benchmark method it annotates.

With the traditional `@Warmup` annotation, which statically pre-defines the number of warm-up iterations, the generated microbenchmark code works as follows: After each warm-up iteration, the microbenchmark checks if the current number of iterations exceeds the configuration specified by the developer. Once this condition is met, the microbenchmark begins collecting performance measurements. The `@DynamicHalt` annotation replaces this static check with a dynamic evaluation using a TSC model. At each iteration, AMBER invokes the `jpt-service`, asking the TSC model to predict whether the current set of measurements is *stable* or *unstable*. If the measurements are deemed *stable*, the microbenchmark dynamically halts the warm-up iterations and begins collecting performance measurements. Otherwise, it runs an additional warm-up iteration and repeats the process.

We implement the `jpt-service` as a Flask[1] web app deployed in a Docker[2] container, which exposes a RESTful API to invoke TSC pre-trained models, *i.e.,* FCN, OSCNN, and Rocket, as described in Sections II-A and II-B. The `@DynamicHalt` annotation allows users to choose the desired model via a parameter, e.g., `@DynamicHalt(model="OSCNN")` specifies the OSCNN model.

```
@Warmup(iterations = 500)
@Measurement(iterations = 100)
public class FlattenRangePerf {
    ...
    @Benchmark
    public void observable(Blackhole bh) {
        observable.subscribe(new PerfConsumer(bh));
    }
}
```

Listing 1: A sample developer's microbenchmark annotation for warm-up configuration.

[1]https://flask.palletsprojects.com
[2]https://www.docker.com

```
@DynamicHalt(model = "OSCNN")
public class FlattenRangePerf {
    ...
    @Benchmark
    public void observable(Blackhole bh) {
        observable.subscribe(new PerfConsumer(bh));
    }
}
```

Listing 2: Microbenchmark annotation featuring AMBER.

IV. USAGE TEMPLATES

This section outlines the steps required to use AMBER. The AMBER setup instructions are provided in the README file of our online appendix[3] and the video tutorial[4].

Once the AMBER configuration is completed, the primary step to enable AMBER dynamic halt is to replace the JMH `@Warmup` and `@Measurement` annotations with the `@DynamicHalt` annotation. Listings 1 and 2 represent practical examples of a microbenchmark from the *RxJava* library[5] using the original and replaced annotations, respectively. As introduced before, the classification algorithm in AMBER can be specified using the `model` element in the `@DynamicHalt` annotation. For instance, in Listing 2 we selected the OSCNN algorithm. Optionally, it can also indicate the host and port exposing the dockerized `jpt` service in the annotation. Similarly, all the parameters can be specified by command line arguments. Once the halt has been configured, the next step is building and running the benchmark classes.

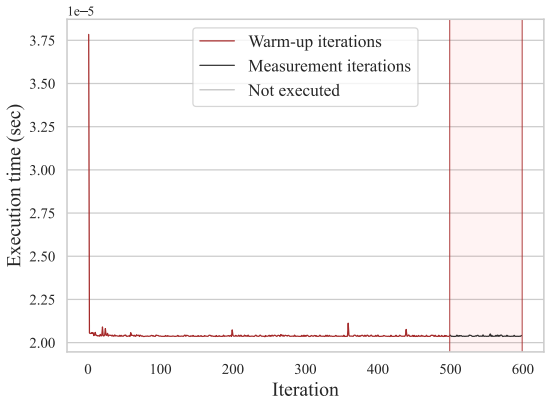
As in JMH, the final report containing the number of warm-up iterations and measurements can be exported in JSON format using the `-rf json` command. The final report provides important insights concerning the amount of warm-up iterations performed using the dynamic halt.

Fig. 3 presents an example of dynamic halting using AMBER, compared to the developer warm-up configuration for the microbenchmark shown in Listings 1 and 2. Specifically, the Fig. 3b highlights the testing time saved when applying the dynamic halting of AMBER.

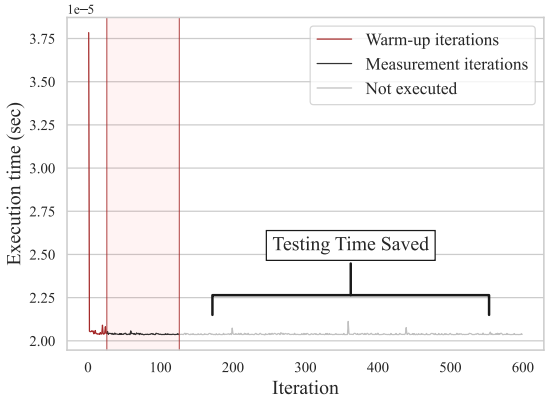
V. EVALUATION SUMMARY

In a previous work [16], we validated and experimented with the TSC algorithms and compared them with the state-of-practice (SOP) and state-of-the-art (SOTA) approaches. First, we validated the suitability of TSC algorithms (*i.e.,* FCN, OSCNN, and ROCKET) in classifying *stable* and *unstable* performance measurements. Then, we conducted an empirical evaluation by comparing the algorithms with the state-of-practice (SOP) and state-of-the-art (SOTA) approaches. Please consider that AMBER has been implemented and evaluated in three distinct versions, each integrating a specific classification algorithm. We assessed the effectiveness of each approach in

[3]https://github.com/AntonioTrovato/AMBER
[4]https://youtu.be/7zOngDQ1z_k
[5]https://github.com/ReactiveX/RxJava.git



(a) Developers warm-up configuration.



(b) AMBER dynamic halt.

Fig. 3: Benchmark execution through (a) a developer's warm-up configuration and (b) the AMBER dynamic halt using OSCNN.

terms of both *results quality* and *testing time*. Specifically, concerning *results quality*, we aimed to assess whether the gathered measurements significantly deviate from the ground-truth steady-state values derived following the approach presented in [2]. Regarding the *testing time*, we analyzed the total time required to execute the microbenchmarks across all the approaches. The analysis used an extensive JMH performance measurements dataset [5], involving time series extracted from 586 JMH microbenchmarks across 30 popular Java Open-Source software systems. In the following, we summarize the achieved results in each research question.

a) RQ_1: To what extent can TSC models accurately classify stable and unstable measurements?: Results depicted a good prediction accuracy in leveraging TSC models to classify *stable* and *unstable* time series segments, with F1-scores ranging between 0.748 and 0.867. Specifically, we noticed that neural network-based models (*i.e.,* FCN and OSCNN) demonstrated lower F1-scores but achieved higher balanced accuracy compared to ROCKET that, despite having a higher recall rate, predicted more false positives.

b) RQ_2: How does AMBER *compare to the state-of-practice (SOP) in Java microbenchmarking?:* In this analysis, we compared the effectiveness of AMBER with the SOP, which halts the microbenchmarks execution after the fixed number of warmup iterations defined beforehand by the developers. Concerning results quality, we observed an evident improvement when employing AMBER equipped with network-based TSC, namely +25.3% for FCN and +27% for OSCNN. In contrast, when using ROCKET, we observed only 9.4% improvements and 25.9% regressions, ascribable to its higher false positives rate. Regarding the testing time, AMBER significantly reduced the testing duration compared to the SOP. Specifically, the SOP approach overestimates the end of the warmup phase in 63.9% of cases, while our framework overestimates it in only 38.1% (FCN), 41.2% (OSCNN), and 13% (ROCKET) of the cases. Overall, the results translate into a net improvement in either results quality or testing time in up to +27% of microbenchmarks, with OSCNN demonstrating the highest net improvement.

c) RQ_3: How does AMBER *compare to the state-of-the-art (SOTA) in Java microbenchmarking?:* We compared AMBER with all three variants proposed by Laaber *et al.* [15] to dynamically stop warmup iterations, namely COV, RCIW, and KLD. Results demonstrated that, although ROCKET achieved lower results quality than SOTA, neural network-based models generally outperformed the SOTA approaches regarding results quality, providing up to +35.3% of improvements. Additionally, we observed that FCN and OSCNN reported improved testing time for about half of the microbenchmarks. In summary, when employing AMBER equipped with neural network-based TSC models, such as FCN and OSCNN, we achieved 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 (20-28.8%), thus resulting in substantial net improvements. Overall, AMBER equipped with OSCNN achieved the best results, leading to a net improvement in up to +35.3% of microbenchmarks.

VI. LIMITATIONS AND EXTENSION

A current limitation of AMBER is that it exclusively supports a specific combination of measurement iterations and time. Specifically, when the microbenchmark is configured with the `@DynamicHalt` annotation, the number of measurement iterations and the measurement time are automatically set to 100 iterations and 100 milliseconds, respectively. A potential extension for AMBER could involve supporting additional measurement configurations, including preprocessing measurements at runtime to align with TSC model input formats or training alternative models that support measurement segments of varying sizes.

In the current AMBER implementation, `jpt-service` is developed as an independent external service and deployed as a Docker container. Future versions could reimplement `jpt-service` in Java, transforming AMBER into a fully self-contained tool and potentially streamlining its adoption.

Researchers and practitioners can extend AMBER to support additional, more effective TSC algorithms, which can be easily achieved by implementing additional RESTful endpoints and the corresponding supported model parameter values for the `@DynamicHalt` annotation.

Regarding the limitation of the empirical evaluation of AMBER (e.g., threats to validity), we refer the reader to our previous paper [16].

VII. Conclusion

This paper presents AMBER, an AI-driven tool designed to dynamically halt warm-up iterations at run-time. Empirical results demonstrate that AMBER improves result quality or significantly reduces testing time across state-of-practice and state-of-the-art techniques. Integrated as an extension of JMH, AMBER ensures seamless compatibility with any existing JMH microbenchmarking suite, making it readily accessible to researchers and practitioners.

VIII. Data Availability Statement

The code and data of this tool demo are permanently stored in Zenodo [22].

References

[1] 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, pp. 373–384. [Online]. Available: https://doi.org/10.1145/3030207.3030213

[2] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133876

[3] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 57–76. [Online]. Available: https://doi.org/10.1145/1297027.1297033

[4] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 63–74. [Online]. Available: https://doi.org/10.1145/2491894.2464160

[5] L. Traini, V. Cortellessa, D. D. Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in java software: are we there yet?" *Empir. Softw. Eng.*, vol. 28, no. 1, p. 13, 2023. [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x

[6] (2014) Java microbenchmark harness (jmh). [Online]. Available: https://github.com/openjdk/jmh/

[7] 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

[8] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 132–143. [Online]. Available: https://doi.org/10.1145/2970276.2970346

[9] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1452–1467, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/7966039/

[10] H. Samoaa and P. Leitner, "An exploratory study of the impact of parameterization on jmh measurement results in open-source projects," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 213–224. [Online]. Available: https://doi.org/10.1145/3427921.3450243

[11] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa, "How software refactoring impacts execution time," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3485136

[12] M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang, and W. Shang, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9813593/

[13] 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

[14] C. Laaber, T. Yue, and S. Ali, "Evaluating search-based software microbenchmark prioritization," *IEEE Trans. Softw. Eng.*, vol. 50, no. 7, p. 1687–1703, Mar. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3380836

[15] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 989–1001. [Online]. Available: https://doi.org/10.1145/3368089.3409683

[16] 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

[17] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 1578–1585. [Online]. Available: https://ieeexplore.ieee.org/document/7966039

[18] W. Tang, G. Long, L. Liu, T. Zhou, M. Blumenstein, and J. Jiang, "Omni-scale CNNs: a simple and effective kernel size configuration for time series classification," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=PDYs7Z2XFGv

[19] A. Dempster, F. Petitjean, and G. I. Webb, "Rocket: exceptionally fast and accurate time series classification using random convolutional kernels," *Data Mining and Knowledge Discovery*, vol. 34, no. 5, pp. 1454–1495, 2020. [Online]. Available: https://doi.org/10.1007/s10618-020-00701-z

[20] Z. Cui, W. Chen, and Y. Chen, "Multi-scale convolutional neural networks for time series classification," *CoRR*, vol. abs/1603.06995, 2016. [Online]. Available: http://arxiv.org/abs/1603.06995

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

[22] (2025) Zenodo replication package of amber. [Online]. Available: https://doi.org/10.5281/zenodo.14749983