UNIVERSITY OF L'AQUILA
DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE AND MATHEMATICS

DOCTORAL THESIS

AI-Driven Software Performance Engineering

Candidate:
Federico DI MENNA

Advisor:
Prof. Vittorio CORTELLESSA

Co-Advisors:
Ing. Maurizio LUCIANELLI
Dr. Luca TRAINI

Doctoral Program Coordinator:
Prof. Davide DI RUSCIO

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in

Information and Communication Technology
Curriculum: Emerging computing models, software architectures, and intelligent systems

XXXVIII Cycle

SSD INF/01

A.Y. 2024/2025

Abstract

Federico DI MENNA

AI-Driven Software Performance Engineering

Delivering efficient software systems is a pivotal requirement in today's software engineering landscape, with significant implications for financial revenues, market competitiveness, and user digital experience. Software Performance Engineering (SPE) provides a well-established set of practices for rigorous performance assurance throughout the entire software lifecycle, from early development to post-deployment. However, as system complexity continues to grow, this task becomes increasingly challenging. By leveraging large-scale data and advanced predictive models, AI-driven approaches have demonstrated their effectiveness in a variety of software engineering tasks, emerging as a promising opportunity to also enhance performance engineering practices.

This thesis reports a set of contributions addressing the challenges that we have afforded in integrating artificial intelligence techniques to support software performance engineering in three crucial phases of the software life-cycle: *development* time, *testing* time and *post-deployment* maintenance effort. During software development, the code optimization task often requires a deep understanding of the code execution behavior at run-time. In this regard, we provide an in-depth investigation on the impact of introducing run-time execution information into language models for code optimization. Experiments with execution-aware versions of CodeT5+ show only limited gains over the standard model. While performance testing, achieving a reasonable trade-off between result quality and testing time is pivotal for realizing effective and practical tests. In that, we introduce an AI-based framework to dynamically detect the end of the Java warm-up phase. The method improves warm-up estimation accuracy, leading to better result quality or reduced testing time compared to existing techniques. After deployment, continuous monitoring is vital to sustain ongoing system performance and stability, typically requiring the analysis of time series data. Within this scope, we investigate the effectiveness of time series forecasting methods applied to software runtime metrics, under varying metric characteristics and forecasting horizons. More in general, time series analysis is widely employed across software testing and post-deployment activities, particularly to assess and improve software quality. In that, we present MALIAS, a meta-learning framework which leverages time series features to automatically select and run the best-performing algorithm for a given software time series, proving its effectiveness in three distinct tasks (i.e., forecasting of software telemetry data, anomaly detection, and steady-state detection in performance testing). As last contribution, we report a practical experience conducted in an industrial environment, showing how can AI-based approaches be successfully integrated for assisting the daily practice for regressions analysis of user digital experience.

Contents

List of Figures

List of Tables

Author's Publications

- Federico Di Menna, Luca Traini, Gabriele Bavota, and Vittorio Cortellessa *Investigating Execution-Aware Language Models for Code Optimization*. Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025. (pp. 204-215). [1] (Chapter 2)
- Luca Traini, Federico Di Menna, and Vittorio 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, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024 (pp. 443-454). ACM. [2] (Chapter 3)
- Antonio Trovato, Luca Traini, Federico Di Menna, and Dario Di Nucci. *AMBER: AI-Enabled Java Microbenchmark Harness*. IEEE Conference on Software Testing, Verification and Validation, ICST 2025 (Testing Tools and Data Showcase Track), Napoli, Italy, March 31 - April 4, 2025 (pp. 762-766). IEEE. [3]. (Chapter 3)
- Federico Di Menna, Luca Traini, and Vittorio Cortellessa. *Time Series Forecasting of Runtime Software Metrics: An Empirical Study*. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024 (pp. 48-59). ACM. [4] (Chapter 4)
- Federico Di Menna, Vittorio Cortellessa, Maurizio Lucianelli, Luca Sardo, amd Luca Traini. *RADig-X: a Tool for Regressions Analysis of User Digital Experience*. In IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024 (Industry Track), Rovaniemi, Finland, March 12-15, 2024 (pp. 359-370). IEEE. [5] Best Industrial Paper Award (Chapter 6)

Submitted Manuscripts

- Federico Di Menna, Luca Traini, and Vittorio Cortellessa. *Forecasting Software Runtime Metrics: A Comparative Study of Classical Statistical, Neural Network, and Foundation Models*. 2025. Submitted to *Journal of Systems and Software*. (Chapter 4)
- Federico Di Menna, Luca Traini, and Vittorio Cortellessa. *Meta-Learning for Time Series-based Tasks in Software Engineering*. 2025. Submitted to *IEEE Transactions on Software Engineering*. (Chapter 5)
- Antonio Trovato, Luca Traini, Federico Di Menna, and Dario Di Nucci. *AMBER: an AI-enabled Java Microbenchmark Harness Extension to Dynamically Terminate Warm-up Iterations*. 2025. Submitted to *Science of Computer Programming* Journal. (Chapter 3)

Chapter 1

Introduction

Performance is a critical non-functional aspect of modern software systems [6], [7], [8]. Optimizations in software efficiency can yield significant economic benefits in industrial settings [9], [10], [11], while also markedly shaping the end-user experience and long-term retention [12], [13], [14], [15]. In this regard, Software Performance Engineering (SPE) provides a systematic, quantitative approach for building software systems that meet performance requirements [16], [17], [18], [19]. This process necessitates rigorous performance assurance activities throughout the entire software lifecycle [20], from early development, where the system is designed and implemented [21], [22], [23], [24], to testing [25], [26], [27], [28] and post-deployment validation [29], [30], [31], [32].

As system complexity continues to grow, this task becomes increasingly challenging [33], [34], [35], [36], [37]. Indeed, modern software systems exhibit an impressive level of complexity, scale, and dynamism, driven by rapid technological advances in computing infrastructure and continuous deployment practices [38], [39], [40]. Additionally, today's software systems evolve continuously [41], increasing the risk to compromise software quality aspects [42]. For instance, recent studies suggest that seemingly non-invasive software changes, such as maintenance activities, can unexpectedly compromise software quality [42], [43]. Conversely, the resources—particularly time—allocated for quality assurance activities are often limited [33]. To cope with this rising complexity, modern software engineering (SE) relies on continuous data collection and analysis to obtain a comprehensive understanding of system structure, evolution, and behavior [44]. Such analysis encompasses information derived from both static resources, such as mined code repositories [45], [46] and issue tracking systems [47], as well as dynamic metrics captured during runtime [30]. In performance assurance activities, static information from software repositories helps to identify how real-life performance issues are caused and resolved [48], [49], [50], [51]. On the other hand, runtime telemetry data is exploited for obtaining a big picture of the system run-time behavior, enabling for proactive analysis [52], [53], [54], [55], [56] and anomalies identification [52], [55], [57], [58], [59], [60], [61], [62]. Nevertheless, the analysis of such data remains computationally demanding and frequently dependent on manual intervention [5], [63], [64], [65].

In this context, automation is pivotal in accelerating performance assurance activities and reducing the associated costs. Artificial Intelligence (AI) achieved impressive results in a plethora of software engineering tasks [66]. Most prominent applications range from code-related tasks—*e.g.,* code summarization [67], [68], vulnerability detection [69], [70], [71], code completion [72], and code refactoring [73]—to post-deployment analysis—*e.g.,* capacity planning [74], [75] and early identification of faults [76], [77], [78]. Given the impressive results of AI-driven approaches in software engineering context, researchers started to explore their effectiveness

in software performance field, like automated code optimization [79], [80] and AI-enhanced software runtime analysis [29], thereby pointing out their strong potential in this context. These promising results motivate the need for a deeper investigation on how modern AI-driven approaches can be integrated in performance-related activities conducted throughout the entire software development lifecycle.

In light of the presented premises, we formulate the *core idea* of this thesis as follows:

> *AI-driven approaches represent a promising solution to effectively support and automate software performance engineering (SPE) activities, from early development to post-deployment. Shedding lights on their strengths and limitations in this context contributes to understand how these techniques can be successfully applied within SPE.*

Grounded in this idea, we introduce a set of research contributions that critically examine the effectiveness of AI-driven techniques for facilitating and improving SPE activities, covering methods that range from well-established machine learning practices to state-of-the-art language models.

1.1 Research Contributions

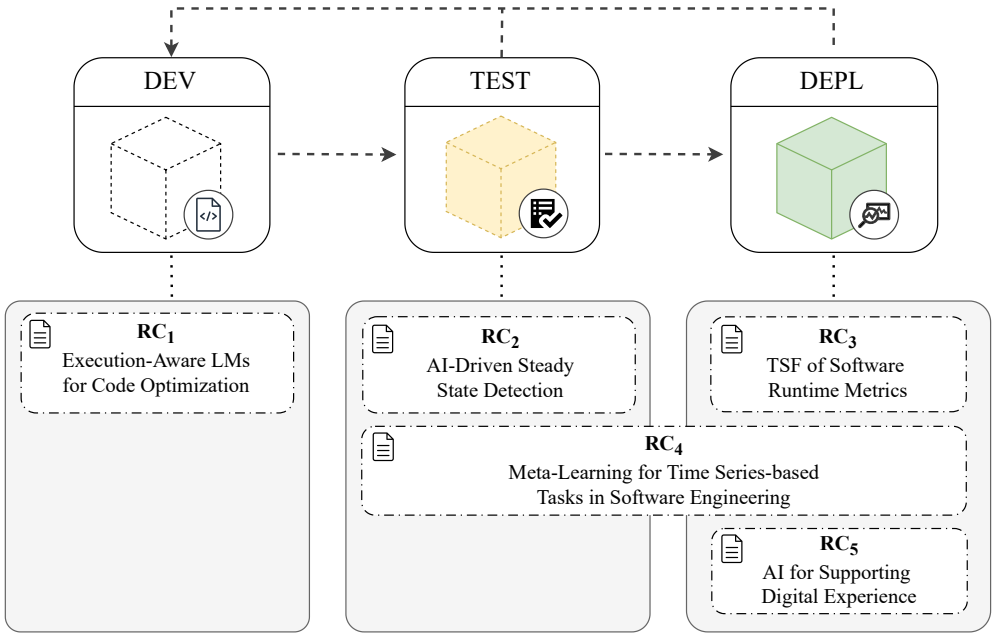


FIGURE 1.1: Process overview of the thesis. The diagram summarizes the consecutive stages described in Chapters 2-6 across the three phases that have been explored—*i.e.*, *DEV, TEST, DEPL*—, and it highlights the corresponding contributions (RC_s).

Figure 1.1 outlines the organization of the five contributions in this thesis, spanning across three phases of the software life-cycle: (i) the **development stage** (labeled *DEV* in the figure), (ii) the **testing stage** (denoted as *TEST*), (iii) and the **post-deployment stage** (indicated as *DEPL*). For each phase, we experiment the adoption of AI-driven approaches in that context. We start by investigating *execution-aware*

language models for code optimization (RC_1), at development stage; then we propose *AI-driven steady state detection in performance testing* (RC_2), followed by an empirical study on *time series forecasting of software runtime metrics* (RC_3), which belongs to the post-deployment stages. We subsequently present a contribution based on *meta-learning for time series-based tasks in software engineering (SE)* context (RC_4) that spans both testing and post-deployment stage, and conclude with a tool showcasing the use of *AI for supporting digital experience* (RC_5) in an industrial context, belonging to post-deployment stage. In the following, we introduce each of the contributions.

1.1.1 Execution-Aware Language Models for Code Optimization

An essential prerequisite for building software systems that fulfill performance requirements lies in the implementation of efficient code. The impressive potential demonstrated by language models in supporting code-related tasks [66], [67], [81], [82], [83] pushed the research community to experiment these models for *source code optimization* [79], [84], [85], [86]. The typical procedure consists of feeding the model with performance-improving code edits and let the model learn the underlying optimization patterns. Yet, this procedure leverages *static* source code representation, while recent studies have shown that the integration of execution information—such as branch coverage and variable values—into the training process can improve the model's ability to perform code-related tasks [70], [87], [88], [89]. Despite these efforts, the implications of introducing code execution awareness into language models for carrying out code optimization have yet to be investigated. Given the close relationship between the run-time execution behavior and the code efficiency [90], we introduce the first research contribution:

CONTRIBUTION

RC_1: *We present an empirical study evaluating the impact of learning code execution aspects on the effectiveness of language models for automated code optimization.*

The first contribution is presented in Chapter 2. It consists of an empirical study where we evaluate and compare four training objectives related to code execution aspects through a set of experiments conducted with a CodeT5+ model [91]. Our analysis reveals that execution-aware versions of the model demonstrated limited capacity to generate semantically correct code compared to the original model, even though they often produce syntactically correct programs. However, when focusing the analysis on correctly optimized code, execution-aware models have the potential to deliver more significant performance improvements. These findings motivate future research to explore additional code execution aspects, leverage other (larger) language models, and experiment with alternative training strategies.

1.1.2 AI-Driven Steady State Detection in Performance Testing

In SPE practice, performance testing is pivotal to uncover performance bugs that might deteriorate software efficiency [92], [93]. A commonly employed method for performance evaluation is *microbenchmarking* [94], consisting in repeatedly executing a small portion of code while gathering execution time measurements. The overall complexity of this practice is mainly related to (i) the *variability* of performance measurements [95], [96], [97], that affects the results quality, and (ii) the practical testing-time constraints [98], [99], [100], [101], [102], [103]. Finding the optimal balance

of results quality and testing time is further complicated when software behavior might drastically change at runtime, a typical feature observed when dealing with JIT (Just In Time)-enabled virtual machines [104], [105], [106]—particularly the Java Virtual Machine (JVM). Indeed, in this case it is difficult to distinguish the initial so-called *warm-up iterations*, where the measurements are still unstable, and the subsequent stable measurements to consider (*steady-state performance*). In this context, halting the testing process as soon as measurements are stable represents the main goal [102], [107], [108]. Traditional approaches either statically fix the number of initial *warmu-up* iterations to ignore (state-of-practice) [102], or use dynamic halting approaches based on statistical heuristics [102], [107], [109] (state-of-the-art).

In this scenario, we re-formulate the steady-state detection (SSD) challenge as a classification problem where a window of consecutive performance measurements have to be classified as *unstable* or *stable*. This can be naturally framed as a time series classification task, a setting commonly explored in the machine learning community, where cutting-edge AI-based classifiers can be employed to draw this distinction. Following this idea, we introduce the second research contribution:

CONTRIBUTION

RC$_2$: *We present an AI-based approach to optimally trade-off between result quality and test execution time in performance testing.*

This second contribution is described in Chapter 3, proposing an AI-based framework to dynamically stop warm-up iterations at run-time. The framework leverages Time Series Classification (TSC) techniques to predict whether a set of measurements is stable or not, dynamically determining the end of the JVM warm-up phase. Our evaluation—conducted over 586 JMH [110] microbenchmarks across 30 Java software systems—shows that the framework delivers significant accuracy improvements of the warm-up estimates compared to both *state-of-practice* and *state-of-the-art* methods. Such results confirm the applicability of AI-driven methodologies in Java performance testing and motivate their further exploration in this context. Along with the conceptual framework [2], we release AMBER [3], an easy-to-use JMH extension that implements our framework.

1.1.3 Time Series Forecasting of Software Runtime Metrics

The characteristics of the production environment may differ from those of the testing settings [57], [111], [112], [113], potentially affecting whether the application is properly operating [114], [115]. Moreover, the continuous release of software changes [41], even multiple times per day [116], [117], make it essential to ensure that software quality is not compromised [55], [118], [119], [120], [121]. To address these challenges, SPE involves post-deployment monitoring activities, where a significant amount of runtime metrics is continuously ingested through Application Performance Monitoring (APM) tools over time [30]. Within the post-deployment stage, we allude to the performance aspects of software systems from a broader perspective, closely related to the software quality perceived by the end user. Indeed, in addition to performance runtime metrics such as response times, we include system and application crashes, as they have an high impact on user experience [122], [123], [124], [125], [126].

Software runtime metrics are typically stored as time series data [127], [128]. Time Series Forecasting (TSF), encompassing both statistical and AI-based approaches, represent a promising opportunity to enhance software monitoring activities. Prior studies investigated about the applicability of TSF approaches for supporting software quality assurance tasks, such as capacity planning [74], [75] and reliability prediction [129], [130]. However, there is still little knowledge about how these methods perform on software runtime metrics, which are notoriously difficult to analyze because of their inherent instability [40], [57], [95], [131]. Aiming to bridge this gap, we introduce the third research contribution:

CONTRIBUTION

RC$_3$: *We conduct an empirical study on the effectiveness of Time Series Forecasting (TSF) methods applied to runtime software metrics, under varying metric characteristics and temporal prediction demands.*

Chapter 4 presents the third contribution structured into two consecutive studies. We start with a preliminary investigation [4] (Section 4.2), by conducting an empirical study on four TSF methods (AI-based and non-AI-based) applied to 25 software metrics collected from a real-world IT infrastructure, covering three runtime aspects of software systems. The findings reveal that TSF methods are indeed suitable for forecasting software runtime metrics, highlighting some connections among specific TSF methods and software metrics classes. Subsequently, the chapter reports an advanced study in Section 4.3, where a substantial expansion of the primary investigation is conducted by covering more and diverse runtime aspects and including a new class of forecasting methods, namely the time series foundation models [132], [133]. Results show that foundation models deliver the highest forecasting accuracy on average. However, the analysis shows that there is no single model that consistently outperform other models across all individual time series, showing the lack of a universally optimal ("*silver bullet*") model that consistently surpasses others in every context [134].

1.1.4 Meta-Learning for Time Series-based Tasks in SE

More in general, time series analysis is widely employed across a variety of software quality assurance activities, such as resource capacity planning [74], [75], [135], [136], software self-adaptation [29], [56] and more [53], [129], [130]. In this respect, one fundamental challenge lies in the selection of the most appropriate algorithm for the target time series data, as also evidenced by RC$_3$ results. In SE domain, the high variability of diverse software runtime metrics (*e.g.*, response times) [95], [131] introduces additional complexity. In this context, *meta-learning* approaches represent a promising solution for tackling this issue. While such approaches have demonstrated effectiveness across several time series domains [137], [138], [139], [140], their efficacy within SE remains largely underexplored. Aiming to fill this knowledge gap, we present the fourth research contribution:

CONTRIBUTION

RC$_4$: *We propose a meta-learning approach for supporting time series–based SE tasks that are critical for system quality assessment.*

As illustrated in Figure 1.1, RC_4 is not limited to post-deployment stage but extends to testing phase, where time series analysis can also be exploited for SSD. Chapter 5 outlines this contribution by presenting a meta-learning framework, namely MALIAS, for time series-related tasks in software engineering context. Specifically, the chapter covers time series forecasting and anomaly detection of runtime software metrics, and so falling under post-deployment analysis, and for time series classification employed in performance testing (SSD). We evaluate MALIAS across the three SE tasks using six publicly available datasets of real-world software time series. Results show that MALIAS consistently outperforms both the best-performing models and two well-established AutoML frameworks across all considered tasks. This improvement can be attributed to MALIAS ability to automatically identify the most suitable time series analysis algorithm given the specific characteristics of the data. However, several challenges emerged concerning the generalization of predictive capabilities across datasets of different domains, indicating that it cannot work as a pre-trained approach in a zero-shot setting, rather it requires domain-specific training for optimal performance.

1.1.5 AI-Assisted Regressions Analysis of User Digital Experience in an Industrial Context

The last part of this thesis contains the research study conducted in the industrial context during the doctoral program. The experience took place in Micron, a leader company in memory solutions manufacturing. In this context, software runtime metrics are continuously recorded across thousands of digital devices used by employees for ensuring a smooth operability. Each software update or software/hardware configuration change poses a potential source of issues that can impact software quality, and consequently the digital experience of employees. However, exploiting the collected data for diagnosis and resolution of issues remains challenging, especially considering the huge amount of metrics that are continuously collected. In this scenario, AI-driven approaches represent promising solutions that can be combined to traditional monitoring approaches. Accordingly, the fifth research contribution presented in this thesis is the following:

CONTRIBUTION

RC_5: *We showcase how AI-driven approaches can be integrated into a real industrial environment for assisting regression analysis of user digital experience.*

The research conducted during the industrial experience is entirely reported in Chapter 6, aiming to support the regression analysis task in a complex hardware/-software environment, and helping to ensure a smooth user experience of employees. Specifically, we build *RADig-X*, an AI-enabled tool to effectively support *anomaly detection*, *digital experience impact evaluation* and *root cause analysis*. Since its deployment, *RADig-X* has helped to identify problematic software updates that have shown relevant impact on the digital experience of the employees.

1.2 Outline

Along with the methodological and results parts, each chapter is structured with an introduction, background, limitations and conclusion. Specifically, this thesis is structured in the following chapters:

- **Chapter 2** presents our contribution concerning RC_1, focusing on code optimization task carried out during software development stage. It explores the effectiveness of execution-aware language models for automated code optimization.
- **Chapter 3** moves the focus to the testing phase, concentrating on RC_2. We present and evaluate an AI-assisted framework for supporting the steady-state detection problem during microbenchmarking in performance testing. The chapter also includes the description of AMBER, a JMH extension implementing our framework.
- **Chapter 4** reports the investigation of time series forecasting (TSF) methods applied to software runtime metrics, as indicated in RC_3, supporting the performance monitoring task during post-deployment.
- **Chapter 5** cuts across both the testing and post-deployment stages by exploring the effectiveness of meta-learning approaches in algorithm selection for time series analysis in software engineering. Specifically, in RC_4, we investigate runtime software metrics forecasting (TSF), anomaly detection (TSAD) and steady-state performance detection (TSC).
- **Chapter 6** presents the industrial experience during the doctoral studies, investigating the employment of AI for supporting user digital experience at scale (RC_5).
- **Chapter 7** concludes this thesis by summarizing the results, findings and experiences carried out during our work. We also recap limitations and indicate future directions based on the outcomes we achieved.

Chapter 2

AI-Assisted Code Optimization: Investigating the Effectivenss of Execution-Aware Language Models

Improving software performance involves activities across various layers of the software stack, ranging from architectural design [141] to compiler optimization [142]. Among these, source code optimization — the process of refining code by selecting efficient data structures and algorithms — stands out as a key approach for enhancing the efficiency of software systems.

As of today, code optimization tasks largely rest on the shoulders of developers. However, with the rising trend of using AI to automate software engineering tasks [2], [66], [67], [81], [82], [83], recent studies have begun exploring the potential of language models to automate code optimization [79], [84], [85], [86]. In these studies, language models are provided with a non-optimized version of a code snippet (e.g., a function) and tasked with generating an optimized version that improves specific performance properties, such as execution time.

While these efforts have demonstrated the significant potential of language models in automating code optimization [79], [84], [85], [86], these models are typically trained on a "static" source code representation and may therefore lack an understanding of how code executes at run-time [87], [88], [89]. Recent studies have shown that integrating these models with code execution information can significantly enhance their effectiveness across a variety of downstream software engineering tasks [70], [88], [89], [143]. For instance, Ding *et al.* [70] proposed a pre-training strategy to teach language models specific aspects of code execution, such as branch coverage and variable states, demonstrating improvements in tasks like clone retrieval and vulnerability detection. Similarly, Ni *et al.* [88] introduced *NExT*, a method that enables language models to inspect variable states of executed code lines and reason about their execution behavior, resulting in a higher fix rate for program repair tasks. Despite these and other efforts [87], [89], [143], [144], [145], the impact of execution-awareness in automated code optimization remains largely unexplored.

Given the close relationship between run-time execution behavior and code efficiency, this chapter investigates how teaching language models to understand code execution behavior affects their effectiveness in optimizing code. Specifically, we first train a CodeT5+ model [91] with training objectives related to four code execution aspects, namely number of line executions, line coverage, branch coverage, and variable states. We then evaluate and compare the code speed-ups achieved by these execution-aware models against those of the standard CodeT5+ model. Our findings reveal that execution-aware models do not outperform the traditional language model in code optimization. On the contrary, they seem to be less effective

than the standard CodeT5+ model.

The contribution of this chapter are as follows:

- A comprehensive evaluation of twelve execution-aware language models, based on CodeT5+, for code optimization. This evaluation encompasses four different code execution aspects and three distinct strategies.
- A complementary analysis providing insights to guide future research in automated code optimization.
- A replication package to reproduce our findings[1].

Chapter Structure Section 2.1 provides background on automated code optimization and execution-aware language models for code. Section 2.2 describes our study design and research questions. Section 2.3 explains the experimental setup. Section 2.4 reports the results. Section 2.5 presents a qualitative analysis, discusses the results, and describes threats to validity. Section 2.6 concludes this chapter.

2.1 Background and Related Work

2.1.1 Language Models for Code Optimization

Transformer-based language models are a category of deep learning models that have driven significant advancements in natural language processing. In recent years, studies have demonstrated the ability of such models to effectively perform various software engineering tasks [66], [67], [71], [81], [82], [83], [146], [147], [148], [149]. Since code optimization involves generating a refinement of a source code that improves efficiency, it can be directly framed as a code-to-code transformation task (specifically, from inefficient to efficient version). For example, Garg *et al.* proposed *DeepDev-PERF* [79], a transformer-based model pretrained on English and C# source code corpora, and further specialized with performance-improving commits. Their work shows that *DeepDev-PERF* can generate code changes that lead to tangible performance optimizations to various open source C# projects.

Shypula *et al.* proposed a dataset of performance-improving code edits (*PIE*) [84], composed of slow-fast C++ program pairs. They exploit this dataset to evaluate a variety of techniques, including fine-tuning and prompting, for adapting well-known language models, such CodeLLama, GPT-3.5 and GPT-4, for code optimization. Their results show that certain combinations of techniques and models can achieve higher optimizations than human reference. Similarly, Chen *et al.* [80] proposes *SUPERSONIC*, a smaller seq2seq model targeting minor source code modifications for optimization. *SUPERSONIC* is trained on C/C++ slow-fast program pairs leveraging a diff-based input representation. Results show that it outperforms larger language models, such as GPT-3.5-Turbo and GPT-4, on code optimization while retaining significant similarity with the original program. RAPGen [86] leverages a pre-constructed knowledge-base to prompt language models in zero-shot to generate code inefficiencies fix. Gao *et al.* [85] modeled the optimization task with a search-based approach, integrating language models with evolutionary search and outperforming several baselines models such as CodeLlama, Gemini and ChatGPT.

Although these studies have demonstrated the effectiveness of language models in optimizing code, they tend overlook the potential implications of integrating code execution information into the model.

[1] https://github.com/SpencerLabAQ/exec-aware-code-opt

2.1.2 Execution-aware Language Models

Most existing language models for code focus on learning static form of code text, however they may lack a semantic understanding of how code execute at run-time [87], [88]. To mitigate this issue, several approaches have been proposed to integrate language models with code execution information. For instance, Ding *et al.* proposed *TRACED* [70], an execution-aware pre-training strategy to capture run-time execution aspects of code. This strategy pre-trains the language model with the task objective of predicting variable states, and line coverage, forcing the model to reason about code execution behavior. They found that learning code execution behavior, significantly improves the capabilities of language models in both clone retrieval and vulnerability detection tasks.

Another example is *SemCoder* [89], an execution-aware language model that leverages monologue reasoning to learn about local execution effects of individual statements, overall input/output behavior, thereby linking static code with code execution behavior. *SemCoder* shows competitive effectiveness with GPT-3.5-turbo on code generation and execution reasoning tasks, despite being significantly (6.7B parameters). A similar approach, namely *NExT*, was recently proposed by Ni *et al.* [88]. This method teaches the language models to inspect the execution traces of programs (variable states of executed lines) and reason about their run-time behavior through chain-of-thought rationales. *NExT* considerably improves the fix rate of a PaLM 2 model for program repair tasks.

Huang *et al.* [143] proposed *FuzzPretrain*, a method to explore run-time execution information of code revealed by their test cases and embed it into the feature representations of code as complements. *FuzzPretrain* yields significant improvements on the code search task over its language model counterparts trained with only static source code.

Previous research has studied the effectiveness of execution-aware models across various software engineering tasks, such as program repair [88], vulnerability detection [70], clone retrieval [70], code generation [89], and code search [148]. However, despite the clear connection between runtime execution behavior and code efficiency, no prior work has directly examined how incorporating code execution information into language models influences their ability to optimize code. This chapter seeks to address this gap.

2.2 Study Design

The *goal* of this study is to investigate how integrating code execution information into language models impacts their ability to optimize code. We analyze twelve execution-aware models, derived from CodeT5+, that incorporate different combinations of four code execution aspects and three training strategies. Specifically, for each code execution aspect, we create execution-aware models using three distinct training strategies. We then evaluate the code optimizations provided by these execution-aware models and compare them to those produced by the standard CodeT5+ model.

In the following sub-sections, we describe the code execution aspects and training strategies investigated in this study, as well as the research questions we aim to address.

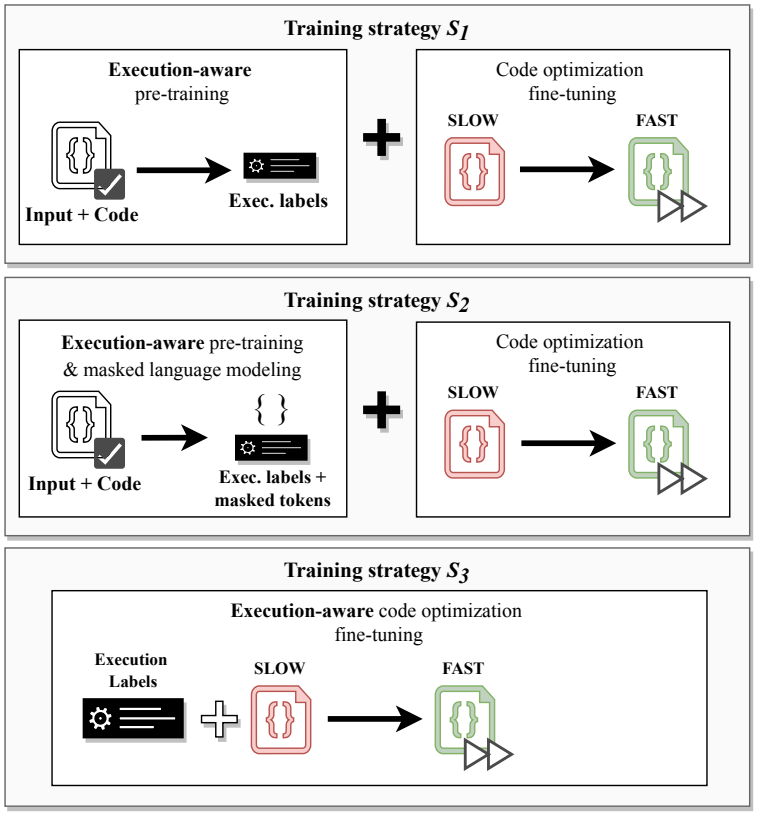


FIGURE 2.1: Overview of the proposed training strategies for building execution-aware language models.

2.2.1 Code Execution Aspects

Line Executions (LE)

This aspect refers to how often a specific line of source code is executed during runtime. Each line of code is assigned a label indicating the number of times it has been executed. This information provides understanding on *hotspots* within the code, *i.e.*, code sections that are frequently executed and may benefit from optimization.

Line Coverage (LC)

Line coverage identifies which lines of source code are executed at least once during execution. This provides insights about which parts of source code are covered after executing the program with a specific input, and which one remain underutilized.

Branch Coverage (BC)

This aspect is primarily designed to determine the parts of the source code associated to a branch. Given the execution of the program with a specific input, each line of code can fall into one of three categories: (i) it is part of a covered branch, (ii) it is part of an uncovered branch, or (iii) it is not part of any branch.

Variable States (VS)

This information focuses on how the code execution changes the state of variables. For example, some variables may control the program's flow (*e.g.*, counters or boolean

flags), while others store intermediate or final results to be returned. Understanding the state of these variables at the end of execution provides insights into the code behavior at runtime.

2.2.2 Training Strategies

We define three distinct training strategies, each applied to a language model which has been already pre-trained on a static representation of code text, specifically CodeT5+. Each strategy is designed to teach the model a particular aspect of code execution using a specific training methodology. Figure 2.1 presents a high-level overview of the stages involved in each training strategy. In particular, we consider the following strategies: (i) *execution-aware pre-training* (S_1), (ii) *execution-aware pre-training* combined with *masked language modeling* (S_2), and (iii) *execution-aware fine-tuning* (S_3). In the following, we provide a description of each training strategy, while their instantiations to specific training datasets are detailed in Section 2.3.1.

S_1 — *Execution-aware Pre-training + Fine-tuning*: This strategy involves two sequential training stages. In the first stage, the model undergoes pre-training to predict aspects of code execution based on specific input test cases. To deepen the model understanding of runtime code execution, we train it on multiple input cases to capture diverse execution behaviors of the same code. We term this process *execution-aware pre-training*. Specifically, the model learns to predict run-time execution information without actually running the code (*i.e.,* it is provided with a program and a specific input and it must predict a specific execution aspect, like for example the line coverage). This step is designed to teach the language model how code executes at runtime. In the second stage, the execution-aware model is fine-tuned for code optimization. This is framed as a sequence-to-sequence downstream task, where the input is a slower version of a program, and the output is an optimized, faster version. Importantly, the optimization process aims to preserve the program's intended functionality; that is, the optimized program must produce the expected output.

S_2 — *Execution-aware Pre-training & Masked Language Modeling + Fine-tuning*: Ding *et al.* [70] demonstrated that combining execution-aware pre-training with *masked language modeling* (MLM) can enhance model effectiveness across multiple software engineering tasks. Building on this approach, this strategy combines an execution-aware objective (as the one described for S_1) with an MLM objective during pre-training. The MLM objective involves randomly masking 15% of tokens in the input sequence and training the model to predict the masked tokens based on the surrounding context. Following the pre-training stage, as in the S_1 strategy, the model is fine-tuned for code optimization.

S_3 — *Execution-aware Fine-tuning*: This strategy does not involve a pre-training stage. Instead, code execution information is incorporated directly into the model input (*i.e.,* the slow version of the code) during fine-tuning. This information is added either line-by-line (for *LE*, *LC*, and *BC*) or at the end of the code (for *VS*). Unlike the first two strategies, this one assumes the availability of an execution trace for the input code. Since code execution information is typically tied to a specific input test case, we adopt different methods depending on the aspect of code execution being considered. For *Line Executions* (*LE*), we use the maximum number of executions recorded for each line across all execution traces. This choice reflects the assumption that these maximum values are more impactful on execution time. For the other code execution aspects, we select a single trace corresponding to a random test case to annotate the slow code.

2.2.3 Evaluation Metrics

To evaluate the model effectiveness in code optimization, we use three evaluation metrics introduced in prior work [84]:

- *Correct (%)*: The percentage of generated programs that successfully execute and produce the expected output for all available test cases.
- *Speedup*: The absolute improvement in terms of execution time. It is defined as $Speedup = \frac{T_I}{T_G}$, where T_I and T_G denote the execution times of the input and generated programs, respectively. In line with previous work [84], we set *Speedup* = 1 when the generated programs are either incorrect or slower than the input program.
- *Percent Optimized (%Opt)*: The percentage of generated code that is both correct and faster across the entire testing set. A generated program is considered faster if it achieves a speed improvement of at least 10%, corresponding to a *Speedup* of at least 1.1x.

2.2.4 Research Questions

We organize the entire analysis pertaining to $\mathbf{RC}_1$ into four research questions, *i.e.*, one for each execution aspect considered in this study:

RQ$_{1.1}$ *How does learning line executions impact the effectiveness of language models for code optimization?* Several code optimizations aim to reduce the frequency of line executions (*e.g.*, loop optimization [150]). This research question investigates the impact of teaching a model to understand line execution behavior on its ability to optimize code.

RQ$_{1.2}$ *How does learning line coverage impact the effectiveness of language models for code optimization?* Here, we study how incorporating line coverage information into the language model affects its effectiveness in code optimization.

RQ$_{1.3}$ *How does learning branch coverage impact the effectiveness of language models for code optimization?* Branch coverage information gives indications of how the conditional control flow statements impacts the code execution at run-time. This information can be crucial to identify potential optimization opportunities [151]. In this research question, we evaluate how learning such information influences the model effectiveness in optimizing code.

RQ$_{1.4}$ *How does learning variable states impact the effectiveness of language models for code optimization?* Having an understanding of the types and values of variables may provide meaningful insights into code execution behaviors, *e.g.*, variables elimination, type optimization and how the instructions reflects on the final values of used variables. Here we analyze whether teaching variable states to language models could enhance their ability to optimize code.

To answer these RQs, we conducted an empirical study to evaluate the effectiveness of twelve execution-aware language models for code optimization. For each code execution aspect (*i.e.*, RQ), we developed three execution-aware models using the training strategies outlined in Section 2.2.2. We then evaluated the effectiveness of these models using the metrics defined in Section 2.2.3 and compared to the standard CodeT5+ model.

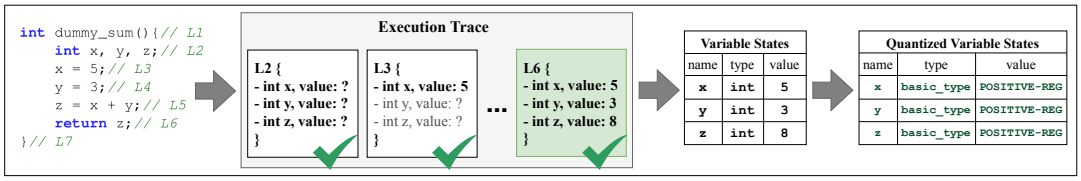


FIGURE 2.2: The figure presents the execution trace corresponding to the outlined sample function, namely `dummy_sum()`. We use the question mark symbol (?) to show that a variable does not yet have an assigned value. Additionally, it also depict the *variable states* information and the associated quantized values.

2.3 Experimental Setup

In this section, we outline a detailed description of our experimental procedure, including the datasets construction, training procedure, baselines and models evaluation.

2.3.1 Datasets Construction

Strategies S_1 and S_2 require an execution-aware pre-training dataset as well as a fine-tuning dataset, whereas the strategy S_3 only needs an execution-aware fine-tuning dataset. We start by describing the data related to the code optimization task and execution information, followed by an explanation of how this data is transformed into datasets tailored to each specific training strategy.

Code optimization

We train language models for code optimization using the *Performance Improving Edits (PIE)* dataset [84], which consists of C++ slow-fast program pairs derived from from IBM *CodeNet* [152]. *CodeNet* is a large-scale dataset of programs, with each program representing a solution to a specific coding problem. It includes multiple programs for each coding problem, which may be authored either by the same developer or by different developers. Additionally, it provides test cases that can be used to evaluate the correctness of each program (based on the problem specifications) and measure its execution time. The *PIE* dataset involve 77k '*slow-fast*' program pairs, where both programs in each pair aim to solve the same coding problem. *PIE* is already provided in a pre-split format consisting of train/validation/test subsets, ensuring that programs belonging to each problem only appear in one of those subsets. Additionally, we apply some preprocessing steps to canonicalize the code and remove any subjective elements in comments and coding style, as outlined in previous work [80]. We use the *gcc* preprocessor to remove comments and *clang-format* to format programs based on the LLVM coding style.

Execution traces

To obtain code execution information, we use the *gdb*[2] debugger to trace program execution, following a method similar to that used in [70]. To prevent prolonged

[2] https://sourceware.org/gdb/

computations, the tracing duration was limited to 500 seconds. The execution tracing procedure may fail if it takes too long or causes an invalid trace due to an unexpected execution interruption (*e.g.*, program crash). The output of the tracing procedure consists of program execution traces that document the complete execution history of each program run. Specifically, such execution traces list all the executed lines, alongside with the name, type, and value of the corresponding program variables at each step of the program execution. We then preprocess these traces to extract execution information needed for our study, such as the execution frequency of each line of code and the state of variables at the end of program execution. For sake of clarity, we depict a sample execution trace referring to a dummy program in Figure 2.2. As illustrated, the trace displays the name, type and value of each variable for every line of code that is run, represented as blocks. The last block in the trace (highlighted in green in the figure) contains the final variable information we use for the *variables state* execution aspect.

Datasets

For fine-tuning the model for code optimization, we directly rely on the *PIE* dataset for both the S_1 and S_2 strategies. Conversely, in execution-aware fine-tuning (*i.e.*, S_3), the dataset must be adapted to incorporate execution information into the *slow* input program. To build the S_3 execution-aware fine-tuning dataset, we performed the tracing procedure over all the programs included in the '*slow*' column of *PIE* dataset. Note that, a '*slow*' program can appear multiple times in the *PIE* training dataset if associated to multiple faster versions. Specifically, among the 21,713 distinct '*slow*' programs contained in the *PIE* training set, 9,145 programs were successfully traced. Similarly, from 978 '*slow*' programs contained in the test set, 466 were traced successfully. This resulted in a total of 31,585 slow-fast training pairs and 466 instances for the testing set.

Differently, to construct the execution-aware pre-training dataset (for S_1 and S_2), we initially excluded all the C++ *CodeNet* programs appearing in either the '*slow*' or '*fast*' columns of the entire *PIE* dataset (43,606 programs). This filter prevents duplicate samples present in both the pre-training and fine-tuning datasets. Additionally, for each coding problem in *CodeNet*, we randomly select up to 150 program-test cases pairs, and we collect their corresponding execution traces. This approach ensures traces collection in a reasonable timeframe, as *CodeNet* comprises more than 8 millions of C++ programs. Indeed, the tracing collection procedure resulted in 119,764 execution traces for building the execution-aware pre-training dataset, involving 87,705 distinct programs. It is important to note that the same program may appear multiple times in this dataset if the corresponding coding problem involves multiples test cases, as each test case generates distinct execution traces.

2.3.2 Quantization Strategy

Language models may face challenges in learning code execution information, as they are primarily continuous numerical values [70]. Let us consider the example of line execution, where each line is associated with the number of times it has been executed for a specific input. This value can be any integer greater than or equal to zero. Thus, it may be challenging for the model to learn the underlying patterns behind this code execution aspect. To address this, as proposed in prior work [70], we define a set of special tokens for each execution aspect to quantize the concrete execution information. For *line executions*, we performed a quantization of concrete

values based on specific ranges, resulting in four unique tokens; <e>: 1 execution, <e+>: $2-5$ executions, <E>: $6-20$ executions, <E+>: $21+$ executions. Lines that are not executed are not marked with any token. The majority of program lines were observed to have zero or one single execution, representing over 80% of all analyzed lines. In light of this, we allocated a dedicated token (<e>) for representing a single execution case. We determined the other thresholds by studying how often lines of code were executed more than once. In particular, we set the higher threshold using the outlier detection rule $T = Q3 + 2.5 * IQR$, and subsequently divided the remaining values into two groups, using the median value as the dividing criterion. The calculated values for median and T were 4 and 23, which we rounded to 5 and 20. To identify the executed lines in the *line coverage* aspect, we only re-use the <e> token; uncovered lines remain unlabeled, as for the *LE* aspect. For learning *branch coverage* we use two tokens, namely <BC> and <BNC>, to indicate that a line is part of a branch that is covered or not covered, respectively. In this case, lines that do not belong to any branch are left unlabeled. Regarding *variable states*, we adapt the quantized representation introduced in TRACED [70] to represent C++ variables, using their name, type and value. In Figure 2.2, we report the variable states quantization step for a dummy example. As depicted, the `int` became `basic_type`, while the concrete values have been converted into `POSITIVE-REG`. We outline all details related to the quantization strategy for each execution aspect in our replication package description.

In Table 2.1, we provide a comprehensive example of the execution information that we feed into the language models. In the first column, we report the traced program alongside with a test case (*i.e.,* program input) and the output obtained from its execution. The other columns outline the special tokens computed for each execution aspect. As show in the table, the *variable states* information refers the entire code execution, whereas the other aspects correspond to line-by-line execution information.

TABLE 2.1: Sample C++ source code aligned with the corrisponding execution information.

C++ Source Code	Line Exec.	Line Coverage	Branch Coverage	Variable States		
				Var. Name	Var. Type	Quantized Val.
Test Case: keyofscience						
#include<iostream>	-	-	-	k	class	OTHER
#include<string>	-	-	-	S	class	OTHER
using namespace std;	-	-	-	s	basic_type	POSITIVE-REG
int main(){	<e>	<e>	-	f	basic_type	POSITIVE-REG
int s=0,f=0;	<e>	<e>	-	i	basic_type	POSITIVE-REG
string S,k="keyence";	<e+>	<e>	-			
cin>>S;	<e>	<e>	-			
for(int i=0;i<S.length();i++){	<e+>	<e>	<BC>			
if(S[i]==k[i])s++;	<e+>	<e>	<BC>			
else break;	<e>	<e>	<BC>			
}	-	-	-			
for(int i=0;i<S.length();i++){	<e+>	<e>	<BC>			
if(S[S.length()-1-i]==k[6-i])f++;	<e+>	<e>	<BC>			
else break;	<e>	<e>	<BC>			
}	-	-	-			
if(s+f>=7)cout<<"YES"<<endl;	<e>	<e>	<BC>			
else cout<<"NO"<<endl;	-	-	<BNC>			
return 0;	<e>	<e>	-			
}	<e>	<e>	-			
Output: YES						

TABLE 2.2: Overview of the total number of instances in the datasets after applying the maximum token limits filter. Note that different execution aspects can lead to dataset sizes, as the number of tokens may be different.

Dataset	BL	LE	LC	BC	VS
Execution-aware Pre-training	-	96,308	96,463	96,463	96,468
Fine-tuning	52,471	52,471			
Execution-aware Fine-tuning	28,753	28,654	28,244	28,654	28,034

2.3.3 Training Procedure

We exploit a Text-To-Text Transfer Transformer (T5) model, namely CodeT5+ [91], which is specifically pre-trained on source code and widely adopted in software engineering literature [68], [153], [154], [155]. With regard to the available computational resources, we selected the CodeT5+ model with a parameter size of 220M. We ran all the experiments using the HuggingFace Transformers library. For all the tasks, we train the model for 1 epoch, with a batch size of 16 and a learning rate of 1e−5. All experiments were conducted using the default AdamW optimizer. The fine-tuning script included in the official CodeT5+ release specifies the maximum token limits for the source and target inputs as 320 and 128, respectively[3]. Since the majority of our instances exceeds these threshold, we decided to increase both the source and target limits to 512 tokens. In Table 2.2, we report the overall instances for each dataset resulting from filtering out the training samples with a number of tokens greater than 512.

In the following, we outline our implementation of each training strategy.

Strategy S_1

Concerning the S_1 execution-aware pre-training, the language model reads a program code and an associated test case (*i.e.*, program input) and generates the corresponding execution information. Let us define the set of the program lines as $L = \{l_1,...,l_n\}$ and the test case as $input$; the model prompt is then built as $I_{PT\_S_1} = \{$`classify:` $+ input +$ `<SEP>` $+ l_1,...,l_n\}$. The corresponding target sequence contains the execution information tokens joined together, separated by the "`\n`" character. Concerning line-wise execution information (*i.e.*, *LE*, *LC* and *BC*), we can define the set of execution information tokens corresponding to the program L as $E = \{e_1,...,e_n\}$ and accordingly build the target sequence as $O_{PT\_S1} = \{e_1,...,e_n\}$. For instance, assuming to deal with *line executions* data, the output sequence corresponding to the example reported in Table 2.1 would be encoded as "`\n\n\n<e>\n<e>\n<e+>...<e>\n<e>`". For the *variable states*, the target sequence consists of a list of inline code comments detailing the name, type, and value of each program variable. For instance, the code reported in Table 2.1 would generate five comments to append to the code, in the following way: "`...// k class OTHER\n...\ni basic_type POSITIVE-REG`". $I_{PT\_S_1}$ and $O_{PT\_S_1}$ are fed into the language model as source and target sequences in a sequence-to-sequence supervised learning setting.

For the S_1 fine-tuning, the code optimization task is modeled as a sequence-to-sequence downstream tasks, in which the model reads the '*slow*' code and generates the optimized version. Assuming to have a slow program composed of n lines ($S = \{s_1,...,s_n\}$) and a faster version composed of m lines ($F = \{f_1,...,f_m\}$), the input sequence for the fine-tuning task is formulated as $I_{FT\_S_1} = \{$`optimize:` $+ s_1,...,s_n\}$. The

[3] https://github.com/salesforce/CodeT5/tree/main/CodeT5%2B

target sequence simply consists of the set of lines of the optimized version joined together: $O_{FT\_S_1} = \{f_1,...f_m\}$. The '`classify:`' and '`optimize:`' prefixes are utilized to assist the language model in distinguishing between the specified tasks.

Strategy S_2

This strategy is similar to strategy S_1, but it integrates the *masked language modeling* (MLM) objective to the execution-aware pre-training process. The model alternates the batches of the execution-aware pre-training dataset for performing both execution-aware and MLM pre-training objectives, aligned with prior research [143]. To distinguish between the two tasks, we use the '`mlm:`' input prefix for MLM batches instances. Formally, let $M = \{m_1,...,m_k\}$ representing the set of k tokens encoding the input program, including those that are masked. The input sequence for MLM is then defined as $I_{PT\_S_2} = \{mlm: + m_1,...,m_k\}$, with the output being the sequence of real tokens corresponding to the masked ones. The instances contained in the batches that are not used for MLM are handled identically to S_1. With this feeding process, we aim to teach the language models to generate both execution and MLM labels. The fine-tuning stage is performed identically to S_1, meaning that $I_{FT\_S_2} = I_{FT\_S_1}$ and $O_{FT\_S_2} = O_{FT\_S_1}$.

Strategy S_3

The strategy S_3 relies to a direct execution-aware fine-tuning process, in which the tokens related to execution information are directly incorporated within the *slow* code, by using code comments. For instance, the 8th line of code in Table 2.1 would be converted into the line `for(int i=0;i<S.length();i++){ // <e+>` in case of *line executions*. For *line executions*, *line coverage* and *branch coverage* aspects, the execution information is generated for each line of code. Then, formalizing the set of source code lines annotated with executions tokens as $S_E = \{se_1,...,se_n\}$, the input sequence ($I_{FT\_S_3}$) is constituted by the flattened version of S_E. Differently, for the *variable states* aspect we generate inline comments for each variable, similarly to S_1, and append them to the '*slow*' program. The target sequence $O_{FT\_S_3}$ corresponds to the optimized version of the code, as for the other strategies.

2.3.4 Baselines

We compare each execution-aware language model against the plain CodeT5+ model directly fine-tuned for the code optimization task (*i.e.*, without injecting any execution information).

In that, we define the following baseline models:

- *Baseline for S_1-S_2 ($BL_{S_{12}}$)*: We train this baseline using the fine-tuning dataset outlined in Table 2.2, involving 52,471 overall instances. Note that the dataset used for training the baseline is identical to the dataset used for fine-tuning the execution-aware models with S_1 and S_2.

- *Baseline for S_3 (BL_{S_3})*: To guarantee a fair comparison regarding strategy S_3, we restrict the analysis to the subset of programs that were successfully traced, applying the same subset for training the baseline, leading to a total of 28,753 instances.

2.3.5 Models Evaluation

We evaluate the effectiveness of each language model using the *PIE* test set [84]. Specifically, we prompt each model to generate an optimized version of each '*slow*' program contained in the test set. Subsequently, the '*slow*' programs and the generated programs are compiled and simulated using the available *CodeNet* test cases on the *gem5* simulator [156]. Through this process, we assess the correctness of the generated programs and measure their execution times, following the methodology outlined in previous work [84]. The *Speedup* for an individual generated program is calculated by averaging the speedups obtained across all the test cases.

The total amount of time required for conducting all the experiments — including execution traces collection, model pre-training, fine-tuning and evaluation — is approximately 3 weeks. The traces collection and evaluation procedures were conducted on a linux server equipped with 40 Intel(R) Xeon(R) 2.30GHz CPUs and 78Gb of RAM, while the models training and inference have been performed over a CentOS HPC cluster equipped with 32 Intel(R) Xeon(R) Gold 6140M CPUs and two Nvidia A100 and A30 GPUs.

2.4 Results

TABLE 2.3: *Correctness*, *Speedup* and *%Opt* scores achieved by each baseline and proposed training strategy in performing code opt of C++ programs. Scenarios that do not include pre-training have been reported in the bottom part of the table for a better understanding. For both the double-staged and single-staged strategies, the highest number in each column is **bolded** and the second-highest is underscored.

Model	Execution Aspect	Training Strategy		Evaluation Metrics		
		Pre-training	Fine-tuning	Correct	Speedup	%Opt
$BL_{S_{12}}$	-	-	Code opt	**18.75%**	**1.79**	**7.68%**
LE_{S_1}	*Line Executions*	Exec-aware	Code opt	12.36%	1.52	5.73%
LE_{S_2}		Exec-aware + MLM	Code opt	11.97%	1.54	5.6%
LC_{S_1}	*Line Coverage*	Exec-aware	Code opt	14.84%	1.7	6.9%
LC_{S_2}		Exec-aware + MLM	Code opt	14.45%	1.76	7.55%
BC_{S_1}	*Branch Coverage*	Exec-aware	Code opt	13.93%	1.67	7.03%
BC_{S_2}		Exec-aware + MLM	Code opt	13.15%	1.55	5.73%
VS_{S_1}	*Variable States*	Exec-aware	Code opt	15.49%	1.69	7.29%
VS_{S_2}		Exec-aware + MLM	Code opt	14.97%	1.65	6.64%
BL_{S_3}	-	-	Code opt	12.06%	2.09	9.22%
LE_{S_3}	*Line Executions*	-	Exec-aware code opt	**12.71%**	**2.11**	**9.35%**
LC_{S_3}	*Line Coverage*	-	Exec-aware code opt	9.97%	1.67	5.84%
BC_{S_3}	*Branch Coverage*	-	Exec-aware code opt	8.65%	1.64	5.76%
VS_{S_3}	*Variable States*	-	Exec-aware code opt	11.55%	1.96	8.35%

Our evaluation follows a common methodology across all the research questions. For each execution-aware and baseline model, we first generate the optimized code and then compute the evaluation metrics defined in Section 2.2.3. Table 2.3 presents the results for the twelve execution-aware models and the two baselines. For the *Speedup* metric, we report the mean *Speedup* across all the programs of the test set.

TABLE 2.4: Comparison of speedups achieved by execution-aware models vs. baselines, evaluated using Wilcoxon test, Vargha-Delaney $\hat{A}_{12}$, and rank biserial correlation (r).

Exec-aware *vs.* Baseline	p-value	$\hat{A}_{12}$	r
LE_{S_1} *vs.* $BL_{S_{12}}$	<0.05	0.446 (N)	-0.643
LE_{S_2} *vs.* BL_{S12}	<0.05	0.467 (N)	-0.399
LE_{S_3} *vs.* BL_{S_3}	0.064	-	-
LC_{S_1} *vs.* $BL_{S_{12}}$	<0.05	0.451 (N)	-0.633
LC_{S_2} *vs.* $BL_{S_{12}}$	<0.05	0.455 (N)	-0.515
LC_{S_3} *vs.* BL_{S_3}	<0.05	0.471 (N)	-0.416
BC_{S_1} *vs.* $BL_{S_{12}}$	<0.05	0.452 (N)	-0.536
BC_{S_2} *vs.* $BL_{S_{12}}$	<0.05	0.447 (N)	-0.61
BC_{S_3} *vs.* BL_{S_3}	<0.05	0.47 (N)	-0.503
VS_{S_1} *vs.* $BL_{S_{12}}$	0.069	-	-
VS_{S_2} *vs.* $BL_{S_{12}}$	0.075	-	-
VS_{S_3} *vs.* BL_{S_3}	<0.05	0.487 (N)	-0.449

The first column of the table specifies the execution-aware or baseline model being evaluated. For instance, LE_{S_2} denotes the execution-aware model trained with line executions information (LE), using the strategy S_2. $BL_{S_{12}}$ indicates the baseline model for strategies S_1 and S_2, while BL_{S_3} denotes the baseline model for S_3.

Additionally, we assess the statistical significance of the differences between the speedup provided by the execution-aware model and baseline model using Wilcoxon's signed-rank test [157], and report two measures of effect size: Vargha-Delaney $\hat{A}_{12}$ [158] and the matched pairs rank biserial correlation r [159]. In this context, the $\hat{A}_{12}$ indicates the proportions of pairs where the *speedup* is higher than the baseline. An $\hat{A}_{12} > 0.5$ indicates that a particular execution-aware model generates more efficient source code than the corresponding baseline. The matched pairs rank biserial correlation r represents the difference between the proportion of favorable and unfavorable evidence. With regard to our work, favorable evidence is represented by a higher speedup for the execution-aware strategy, meaning that an $r > 0$ is denoting that the execution-aware model performs better than the baseline. Results from the comparison employing the Wilcoxon's signed-rank test, along with effect sizes, are reported in Table 2.4. In the following, we discuss the achieved results by research questions.

2.4.1 $RQ_{1.1}$: How does learning line executions impact the effectiveness of language models for code optimization?

By observing Table 2.3, we find that both LE_{S_1} and LE_{S_2} models perform generally worse than the $BL_{S_{12}}$ baseline across all the evaluation metrics. Concerning the correctness of the generated code, we observe lower percentages compared to the baseline, with 12.36% correct programs for LE_{S_1} and 11,97% for LE_{S_2}, versus a correctness of 18.75% for $BL_{S_{12}}$. We also observe that execution-aware pre-training works slightly better for correctness when combined with masked language modeling (*i.e.*, LE_{S_2}). We find that the speedup achieved by the code generated by execution-aware models (1.52x and 1.54x for LE_{S_1} and LE_{S_2}, respectively) is not as good as the one reached by the baseline (1,79x). The *%Opt* values confirm the same trend observed for the other two metrics, with lower percentages of optimized programs for the execution-aware models.

Contrariwise, we find that execution-aware fine-tuning (LE_{S_3}) works slightly better than its baseline BL_{S_3} across all the evaluation metrics. Particularly, LE_{S_3} reports a higher number of correct programs (12.71% vs. 12.06%), a highest speedup (2.11x vs. 2.09x), and a higher percentage of optimized programs (9.35% vs. 9.22). Since these improvements appear marginal, we further investigated the statistical significance of difference between the speedups provided by LE_{S_3} versus those of BL_{S_3}. Table 2.4 shows the results of the Wilcoxon test, which suggest no statistical difference for the speedup, *i.e.*, $p > 0.05$.

Answer to RQ$_{1.1}$: Teaching line execution behavior to the language model does not enhance its code optimization capabilities. Execution-aware fine-tuning shows a slight improvement, but without statistical significance.

2.4.2 RQ$_{1.2}$: How does learning line coverage impact the effectiveness of language models for code optimization?

Results for line coverage reveal a trend similar to that observed in RQ1, with baselines outperforming all three execution-aware models. When comparing LC_{S_1} and LC_{S_2}, we observe an increase in speedup and *%Opt* values when the MLM is combined with execution-aware pre-training (1.76x vs. 1.7x for speedup, and 7.55x vs. 6.9x for *%Opt*), despite a slight reduction in correctness (14.45% vs. 14.84%).

By observing Table 2.3, we notice that, compared to other execution-aware models based on the strategies S_1 and S_2, line coverage demonstrates the highest effectiveness in terms of speedup and the percentage of optimized programs. This underlines that the effectiveness of execution-aware training strategies are closely related to the specific execution aspect used to feed the model. For the execution-aware fine-tuning strategy LC_{S_3}, we observe a similar trend, showing lower effectiveness compared to the baseline BL_{S_3}, *e.g.*, speedup of 1.67x vs. 2.11x.

Wilcoxon's test indicates statistically significant differences for all comparisons between execution-aware models and baselines ($p<0.05$), with a negligible disadvantage for all execution-aware models in terms of effect size ($\hat{A}_{12} <0.5$).

Answer to RQ$_{1.2}$: Learning line coverage does not improve the model effectiveness in optimizing code. Execution-aware models perform worse than baseline models across all metrics.

2.4.3 RQ$_{1.3}$: How does learning branch coverage impact the effectiveness of language models for code optimization?

As shown in Table 2.3, execution-aware models consistently underperform compared to their respective baselines across all evaluation metrics. Specifically, we observe reduced correctness, with values of 13.93%, 13.15%, and 8.65% for BC_{S_1}, BC_{S_2}, and BC_{S_3}, respectively. Regarding optimization metrics, execution-aware models also lag behind the baselines, achieving speed-ups of up to 1.67x (compared to 1.79x for $BL_{S_{12}}$ and 2.09x for BL_{S_3}) and *%Opt* values of 7.03% (vs. 7.68% for $BL_{S_{12}}$ and 9.22% for BL_{S_3}).

When focusing exclusively on execution-aware fine-tuning, BC_{S_3} exhibits worse effectiveness across all evaluation metrics compared to the baseline BL_{S_3}.

TABLE 2.5: Percentages of programs that successfully compile, execute, and produce correct outputs.

Model	Compiled	Executed	Correct
$BL_{S_{12}}$	78.65%	76.95%	**18.75%**
LE_{S_1}	81.25%	78.65%	12.36%
LE_{S_2}	80.47%	77.21%	11.97%
LC_{S_1}	85.68%	82.94%	14.84%
LC_{S_2}	74.48%	72.53%	14.45%
BC_{S_1}	80.08%	77.08%	13.93%
BC_{S_2}	**89.19%**	**86.72%**	13.15%
VS_{S_1}	85.03%	82.29%	15.49%
VS_{S_2}	85.68%	80.99%	14.97%
BL_{S_3}	**89.83%**	**88.65%**	12.06%
LE_{S_3}	86.81%	84.65%	**12.71%**
LC_{S_3}	86.13%	84.18%	9.97%
BC_{S_3}	85.10%	84.38%	8.65%
VS_{S_3}	86.73%	84.77%	11.55%

Answer to RQ$_{1.3}$: The integration of branch coverage information into language models reduces both correctness and optimization metrics across the execution-aware training strategies.

2.4.4 RQ$_{1.4}$: How does learning variable states impact the effectiveness of language models for code optimization?

The results for variable states in Table 2.3 indicate a degradation in all evaluation metrics when employing any of the considered execution-aware strategies. Specifically, correctness decreases on average from 18.75% for the baseline $BL_{S_{12}}$ to 15.49% for the execution-aware model VS_{S_1} and further to 14.97% for the VS_{S_2} model. Regarding execution-aware fine-tuning, we observe a similar decline in correctness, from 12.06% ($BL_{S_{12}}$) to 11.55% (VS_{S_3}). Speedup and *%Opt* also follow a similar downward trend compared to the baselines.

Answer to RQ$_{1.4}$: Learning variable states in language models does not enhance the effectiveness of code optimization. On the contrary, we observe reduced correctness and fewer optimizations in all execution-aware models.

2.5 Discussion

We conduct a complementary analysis to better understand the results of our study and to identify potential insights for future research on execution-aware code optimization.

2.5.1 Correctness analysis

Our evaluation reveals the limited capability of execution-aware models to produce correct code, *i.e.,* code that successfully passes all test cases. To better understand

this limitation, we delve deeper into the reasons behind the lack of correctness. In particular, we analyze the proportions of generated programs that: (i) exhibit well-formed syntax (*i.e.,*, the code successfully compiles); (ii) do not result in runtime errors (*i.e.,*, the code successfully executes); and (iii) produce the expected output (*i.e.,*, the code passes all test cases).

Table 2.5 presents the percentages of programs that successfully compile (*Compiled*), execute without errors (*Executed*), and produce the expected output (*Correct*). We observe that execution-aware models exhibit comparable, and sometimes superior, effectiveness in generating programs that successfully compile and execute. Specifically, when considering execution-aware pre-training strategies (S_1 and S_2), these models outperform the baseline $BL_{S_{12}}$ in generating programs that compile and execute successfully. For example, LE_{S_1}, LC_{S_1}, BC_{S_2}, and VS_{S_2} generate programs that successfully compile in 81.25%, 85.68%, 89.19%, and 85.68% of cases, respectively, compared to 78.65% for $BL_{S_{12}}$. Similarly, LE_{S_1}, LC_{S_1}, BC_{S_2}, and VS_{S_1} generate programs that successfully execute in 78.65%, 82.94%, 86.72%, and 82.29% of cases, respectively, compared to 76.95% for $BL_{S_{12}}$. However, when considering the correctness of the generated programs, we observe a notable drop in effectiveness for execution-aware models. Models using pre-training strategies S_1 and S_2 achieve a correctness of up to 15.49%, whereas the baseline $BL_{S_{12}}$ achieves a correctness of 18.75%. These results suggest that while execution-aware models generally perform well in generating well-formed code that successfully executes, they may be limited in their semantic understanding of the intended functionality (*i.e.,* correctness). This finding is somewhat counterintuitive, as execution-aware training strategies are typically designed to enhance language models' semantic understanding of code [70], [88], [89]. Nevertheless, in light of these findings, we encourage future research on code optimization to focus on improving the correctness of programs generated by execution-aware language models. This could involve exploring additional aspects of code execution or developing alternative, more effective training strategies.

2.5.2 Speedup analysis

We assess the average speedup across the entire set of generated code, accounting for both incorrect and slower programs. In line with prior research [84], we assign a *speedup = 1* in such cases. To better understand the speedup provided by execution-aware models, here we focus on analyzing only the correct programs generated by the models, as well as the programs that achieve an non-negligible optimization of execution time (greater than 1%). Table 2.6 presents the mean speedup achieved by each model, calculated exclusively for programs that produce the expected outputs (*Correct Programs*) and for those that optimize the input program (*Optimized Programs*). Additionally, we report the number of program instances considered in each case. Execution-aware models deliver comparable or even superior speedup compared to baseline models when considering only correct programs. This trend is more evident in models utilizing execution-aware pre-training (*i.e.,* S_1 and S_2): BC_{S_1} and LC_{S_2} achieve mean speedups of 5.83x and 6.22x, respectively, whereas the baseline $BC_{S_{12}}$ achieves a speedup of 5.2x. Furthermore, when focusing on optimized programs, we observe that execution-aware models achieve significantly higher speedups compared to baseline models. For instance, LC_{S_1} achieves a speedup of 9.96x, while the baseline $BL_{S_{12}}$ 5.29x. Similarly, LC_{S_3} achieves a speedup of 11.67x, compared to 10.3x for the baseline BL_{S_3}. These findings indicate that execution-aware models have the potential to deliver significant performance improvements when they successfully generate correct or optimized programs.

TABLE 2.6: Speedup statistics considering only (i) correct programs and (ii) optimized programs.

Model	Speedup			
	Correct Programs		Optimized Programs	
	Instances	*Mean*	*Instances*	*Mean*
$BL_{S_{12}}$	144	5.2x	141	5.29x
LE_{S_1}	95	5.2x	91	5.39x
LC_{S_1}	114	5.71x	60	**9.96x**
BC_{S_1}	107	5.83x	63	9.21x
VS_{S_1}	119	5.44x	111	5.76x
LE_{S_2}	92	5.49x	90	5.58x
LC_{S_2}	111	**6.22x**	66	9.79x
BC_{S_2}	101	5.21x	55	8.72x
VS_{S_2}	115	5.37x	109	5.61x
BL_{S_3}	51	**10.12x**	50	10.3x
LE_{S_3}	53	9.76x	47	10.87x
LC_{S_3}	41	7.76x	26	**11.67x**
BC_{S_3}	36	8.4x	26	11.24x
VS_{S_3}	47	9.29x	40	10.74x

2.5.3 Threats to Validity

Construct validity

Considering different execution aspects and quantization criteria could produce results that differ from those presented in this study. Nevertheless, we focused on execution aspects that have been successfully employed in prior work [70]. Using alternative training strategies to learn code execution information or larger language models may yield different outcomes. Still, we evaluated a total of twelve execution-aware models by employing three distinct training strategies, and four execution aspects. Additionally, we employed CodeT5+, a language model widely adopted in the software engineering literature [68], [153], [154], [155].

Internal validity

The choice of the language model hyperparameters plays a key role in determining the model effectiveness. We try to mitigate possible biases by adopting the default parameters of the CodeT5+ release package. However, choosing different hyperparameters may lead to different results. Measuring program execution time typically involves a certain degree of variability [95]. To mitigate this risk and improve the scalability of our experiments, similar to previous work [84], we rely on a deterministic simulator, namely *gem5*, to measure program execution time. However, executing programs on concrete hardware enviroments may yield different results. In addition, the reliability of our performance benchmarking is significantly affected by the choice of the test cases, which we directly inherit from CodeNet.

External validity

Our evaluation is based on C++ programs from CodeNet, which primarily consist of self-contained programs to solve specific coding problems. In addition, a random sampling was considered during dataset construction. The findings this study,

they may not generalize to programs written in other programming languages or to more complex software systems involving multiple modules, third-party libraries, or intricate interdependencies.

2.6 Conclusion

We investigate the effectiveness of execution-aware language models in code optimization. We evaluate four execution aspects and three training strategies to teach a CodeT5+ how code executes at runtime. Our findings reveal that learning code execution behavior does not enhance the model's ability to optimize code. A significant weakness of execution-aware models is their limited capacity to generate semantically correct code, even though they often produce syntactically correct programs. We encourage future research to explore additional code execution aspects, leverage other (larger) language models, and experiment with alternative training strategies. This research objective informs and drives our future agenda.

Chapter 3

AI for Performance Testing: Automating JVM Steady-State Performance Detection

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 [95], [96], [97]. This typically involves balancing the need for quality of performance test results against the practical constraints of resources and testing time [98], [99], [100], [101], [102], [103].

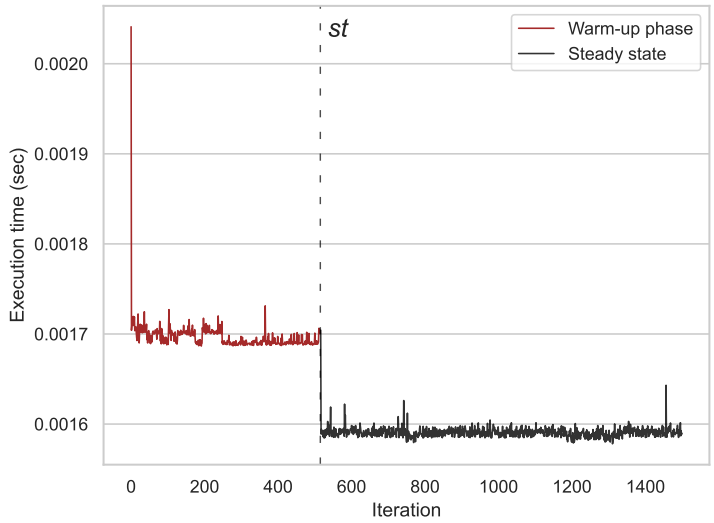


FIGURE 3.1: Execution time of a Java microbenchmark (from the *Roaring Bitmap* project) over consecutive iterations. The execution is characterized by an initial warm-up phase and a subsequent steady-state of performance. The grey dotted line indicates the iteration *st* at which steady-state is attained.

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 [108]. During this phase, the Java Virtual Machine (JVM) performs a wide range of optimizations, leading to fluctuations in performance behavior (as shown in Figure 3.1) that may adversely affect result quality [107], [109]. 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.

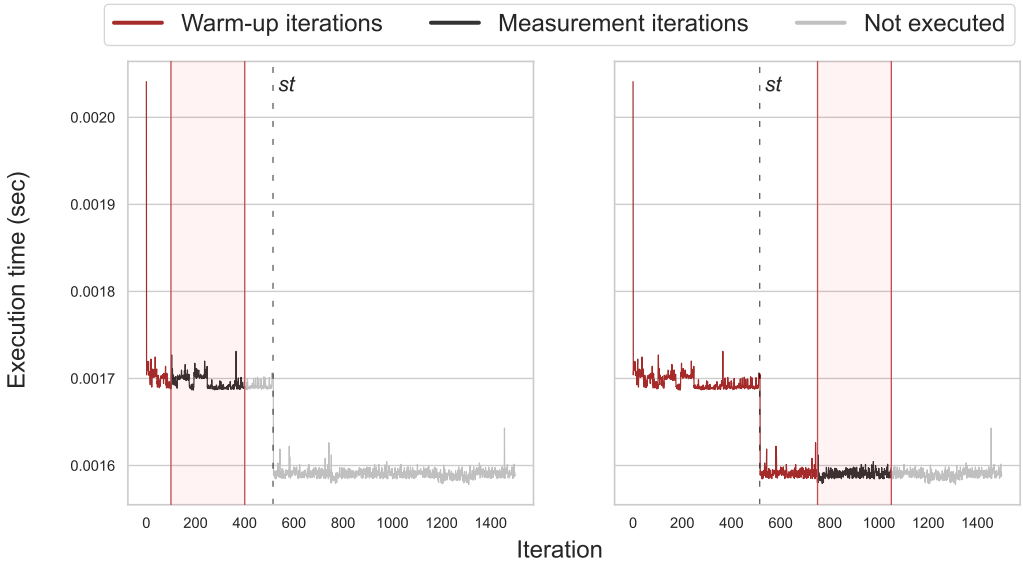


FIGURE 3.2: The left box shows an example of an insufficient number of warm-up iterations, which leads to the collection of measurements unrepresentative of the steady-state. The right plot depicts a scenario with an excessive number of warm-up iterations, which increases the testing time.

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 [33], [160], [161]. 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 [102]. However, studies show that using a fixed number of iterations may misrepresent the warm-up phase [102], [107], [109]. In response to this, researchers developed state-of-art techniques that leverage statistical heuristics to dynamically stop the number of warm-up iterations at run-time [102], [109]. Despite their advantages, these techniques are prone to significant inaccuracies that may either compromise result quality or increase testing time [107]. It still remains unclear how to effectively stop warm-up iterations to ensure result quality in a timely manner.

In this chapter, 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 [94]. Results show that our framework noticeably improves the cost-effectiveness of performance testing.

The main contributions of this chapter 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 pre-trained TSC models to aid future studies[1].

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

LISTING 3.1: Example of a JMH microbenchmark from the *protostuff* Java library. The `@Warmup` annotation is used to define the number of warm-up iterations.

3.1 Background

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

3.1.1 Java Microbenchmarking

Microbenchmarking is a type of unit-level performance testing, commonly used in Java context [94]. 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* [108].

Software engineers employ *warm-up iterations* to ensure that the microbenchmark achieves a steady-state before starting to collect measurements. As shown in Figure 3.2, 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.

[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.

[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 [162].

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) [110], 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 3.1. Nevertheless, prior work has shown that using a fixed number of iterations may often produce suboptimal estimates of the warm-up phase [102], [107], [109].

State-of-the-art (SOTA). Researchers have developed techniques that can dynamically stop warm-up iterations at run-time. Georges *et al.* [109] introduced a first such approach, using a preset threshold on the coefficient of variation [163] to determine the end of the warm-up phase. However, subsequent studies have identified Georges *et al.*'s heuristic as both inaccurate [98] and unrealistic [102], [107] in practical scenarios. Another technique was recently proposed by Laaber *et al.* [102]. 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 runtime. Laaber *et al.* provide three different variants of this technique, based on distinct statistical metrics, namely coefficient of variation (COV) [163],[3] relative confidence interval width [164], [165] (RCIW), and Kullback-Leibler divergence [101], [166] (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 [107].

3.1.2 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 [167], e-health [168], natural disasters [169], and finance [170]. Due to its broad applicability, hundreds of TSC algorithms have been proposed over the last decades [171], [172]. These algorithms span various categories, including distance-based [173], dictionary-based [174], convolutional-based [175], [176], or deep learning approaches [177]. Recently, the latter two categories have demonstrated notable advancements, enabling them to attain state-of-the-art performance on established TSC benchmarks [171], [177], [178], [179], such as the UCR archive [180]. 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 [176], and two deep learning models, namely FCN [181] and OS-CNN [179].

3.2 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]Both the approaches of Georges *et al.* [109] and Laaber *et al.* [102] 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.

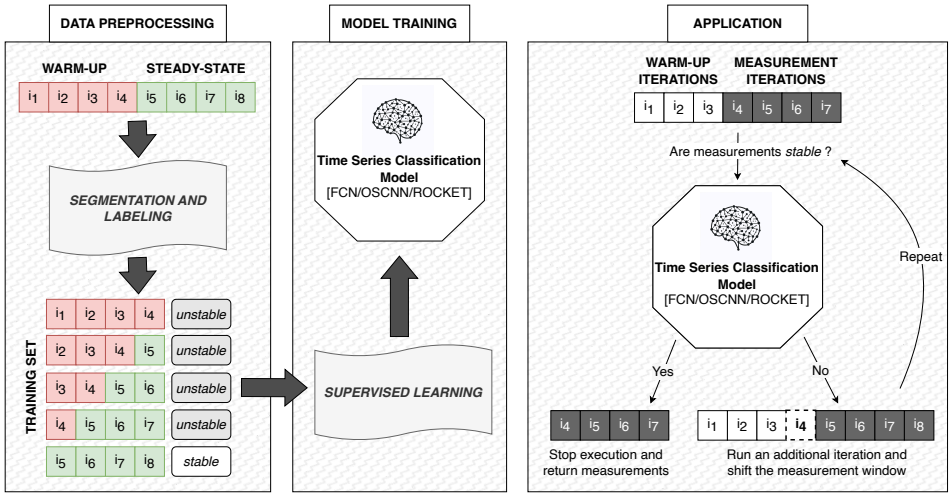


FIGURE 3.3: Overview of the main phases of our framework: ***Data Preprocessing***, ***Model Training*** and ***Application***.

3.2.1 Overview

As shown in Figure 3.3, 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 derive the number of warm-up iterations, thus stopping the testing process (*Application*).

3.2.2 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.* [108]. This notion leverages change point detection, namely PELT [182], 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.

Figure 3.3 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.

3.2.3 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.* [181] for classifying univariate time series. This neural network architecture acts as feature extractor stacking three convolutional layers [183], each followed by a batch normalization layer [184] and ReLU activation layer [185]. The features are then processed through a global average pooling layer [186] and finally passed into a softmax classifier to obtain the final output label. Prior work has shown that this architecture can achieve state-of-the-art performance in TSC [177].

Omni-Scale Convolutional Network (OSCNN) introduces the Omni-Scale block [179], 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 [187].

Random Convolutional Kernel Transform (ROCKET) is a pipeline classifier [176]. 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 [188] 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 [171], [176], [179].

3.2.4 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. Figure 3.3 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.

3.3 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.

3.3.1 Dataset of JMH Performance Measurements

We utilize the dataset by Traini *et al.* [107], which includes performance measurements from 586 JMH microbenchmarks across 30 Java software systems (refer to Table 3.1 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 [108].

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.* [107]). 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.

3.3.2 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

TABLE 3.1: Overview of the Java systems, including the name of each GitHub repository, the number of stars (indicating how many users appreciated the project) 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

(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 [102]. 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 [102]. 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 [107]. 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 [102], 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.

3.3.3 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 [42], [102], [189], [190] by calculating the confidence interval for the execution time ratio [98]. Specifically, we employ the bootstrap method [164], [165] with 10,000 iterations [191], using hierarchical random resampling with replacement at two levels [165], 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.

3.3.4 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 [181]. 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.* [179], which uses PyTorch[6]. We maintained the default hyperparameter settings defined by the authors.

[4]Time series where the steady-state is not reached are excluded.
[5]https://www.tensorflow.org
[6]https://pytorch.org

TABLE 3.2: 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

- **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 [192] 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 [193] 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.

3.4 Results

This section presents the results of this study. We initially assess the effectiveness of TSC models in classifying stable and unstable measurements (**RQ**$_{2.1}$), and subsequently compare our framework with both *state-of-the-art* (**RQ**$_{2.2}$) and *state-of-practice* (**RQ**$_{2.3}$) approaches. In that, we pose and answer the following research questions:

3.4.1 RQ$_{2.1}$: To what extent can AI-driven approaches improve the effectiveness of dynamic reconfiguration in performance testing?

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.

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 3.2 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

[7] https://www.aeon-toolkit.org

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 ($\sim$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$_{2.1}$: 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.

3.4.2 RQ$_{2.2}$: 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 [45], [107]. 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 [157] along with two measures of effect size: the Vargha-Delaney $\hat{A}_{12}$ [158] and the matched pairs rank biserial correlation r [159]. 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.

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).

TABLE 3.3: Results for the WEE comparison of models *vs.* SOP ($RQ_{2.2}$).

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

TABLE 3.4: Percentages of improvement and regression when comparing models *vs.* SOP ($RQ_{2.2}$).

Model *vs.* SOP	**Improvement (%)**			**Regression (%)**			**Net Improvement (%)**
	Results Quality	Testing Time	***Total***	Results 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***

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 3.3 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 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 3.4 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%

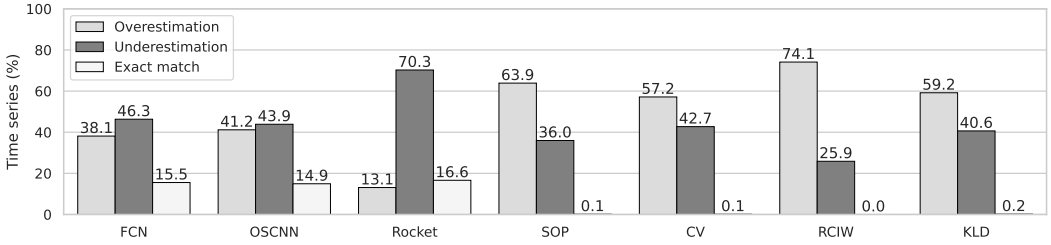


FIGURE 3.4: Percentages of *overestimation*, *underestimation*, and *exact match* for models/SOP/SOTA ($RQ_{2.2/2.3}$).

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 Figure 3.4, 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 $RQ_{2.1}$ 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 3.5. 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 3.4, 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 3.5, 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 Figure 3.4, 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 3.4). 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.

TABLE 3.5: Relative measurement deviation and testing time of models, SOP, and SOTA ($RQ_{2.2/2.3}$).

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)

Answer to $RQ_{2.2}$: 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.

3.4.3 $RQ_{2.3}$: 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.* [102], 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 [102]. As in $RQ_{2.2}$, we employ the Wilcoxon signed-rank test [157], 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 RQ2.2, we discuss the results of this RQ from various perspectives.

Warm-up estimation accuracy. Table 3.6 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 [158]. The rank biserial correlation r ranges from 0.157 to 0.414, suggesting a prevalence for the favorable outcome.

Result quality. Table 3.7 presents the percentages of improvement and regression in result quality and testing time. We observe that ROCKET performs worse than

TABLE 3.6: Results for the WEE comparison of models *vs.* SOTA ($RQ_{2.3}$).

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

TABLE 3.7: Percentages of improvement and regression when comparing models *vs.* SOTA ($RQ_{2.3}$).

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***

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 3.5. 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.

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 Figure 3.4), 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 3.5, 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 3.7, 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 3.7, we can observe that OSCNN provides a net improvement over CV, RCIW, and KLD by +35.3%, +32.9%, and +25.9%, respectively.

Answer to $RQ_{2.3}$: 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.

3.5 AMBER: AI-Enabled Java Microbenchmark Harness

Along with the conceptual framework, we release AMBER (**A**I-enabled Java **M**icro**be**nchmark Ha**r**ness), a tool that implements our AI-driven approach [2] to halt warm-up iterations at run-time dynamically. AMBER relies on Time Series Classification (TSC) algorithms presented above 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. AMBER extends the JMH framework, enabling practitioners and researchers to seamlessly adopt our AI-driven approach for dynamically halting warm-up iterations.

3.5.1 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. JMH is composed of two core modules, namely `jmh-core` and `jmh-generator-annprocess`. We introduce an additional module, namely `jpt-service`, that integrates the ability to dynamically halt warm-up iterations at run-time. 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[8] web app deployed in a Docker[9] container, which exposes a RESTful API to invoke TSC pre-trained models, *i.e.*, FCN, OSCNN, and Rocket. The `@DynamicHalt` annotation allows users to choose the desired model via a parameter, e.g., `@DynamicHalt(model="OSCNN")` specifies the OSCNN model.

This section outlines the steps required to use AMBER. The AMBER setup instructions are provided in the README file of our online appendix[10] and the video tutorial[11].

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 3.2 and 3.3 represent practical examples of a microbenchmark from the *RxJava* library[12] 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 3.3 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.

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

LISTING 3.2: A sample developer's microbenchmark annotation for warm-up configuration.

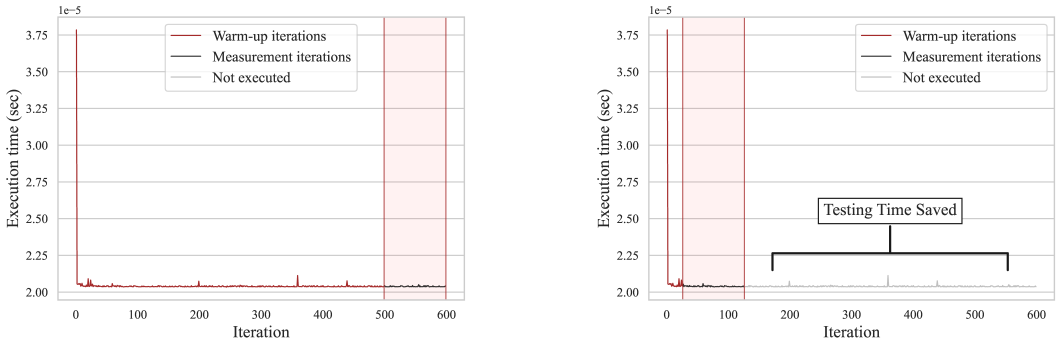
[8] https://flask.palletsprojects.com
[9] https://www.docker.com
[10] https://github.com/AntonioTrovato/AMBER
[11] https://youtu.be/7zOngDQ1z_k
[12] https://github.com/ReactiveX/RxJava.git

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

LISTING 3.3: Microbenchmark annotation featuring AMBER.

3.5.2 Usage Templates

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.



(A) Developers warm-up configuration.

(B) AMBER dynamic halt.

FIGURE 3.5: Benchmark execution through (A) a developer's warm-up configuration and (B) the AMBER dynamic halt using OSCNN.

Figure 3.5 presents an example of dynamic halting using AMBER, compared to the developer warm-up configuration for the microbenchmark shown in Listings 3.2 and 3.3. Specifically, the Figure 3.5b highlights the testing time saved when applying the dynamic halting of AMBER.

3.6 Threats to validity

Construct validity. We derive the number of warm-up iterations through post-hoc analysis using a methodology similar to [102], [107]. 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 3.5. Therefore, they are unlikely to impact our main findings.

We rely on the notion of steady-state as defined by Barrett *et al.* [108], 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 3.1). Furthermore, the number of systems involved in our study is larger than the ones considered in most recent software performance studies [47], [102], [189], [194], [195].

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 [107], 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.

3.7 Related Work

Besides the works discussed in Section 3.1, other techniques have been proposed to estimate the attainment of steady-state in performance tests. Kalibera and Jones [98] 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.* [108] proposed an automated technique based on change point analysis [182] 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.* [103] proposed a technique to stop load tests when performance metric values become repetitive. He *et al.* [101] proposed a statistical approach to halt performance tests in the cloud. Abdullah *et al.* [196] proposed an approach for the early stopping of non-productive experiments in performance testing.

3.8 Conclusion

In this chapter, 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 [197], [198], configuration tuning [167], or software self-adaptation [199]), 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.

Chapter 4

AI for Software Performance Monitoring: Forecasting Runtime Metrics

Given the increasing complexity of today's software systems, monitoring has become an essential activity for ensuring quality of software at runtime. Modern software systems are subject to continuous changes [41] and handle time-varying mixture of load [131], which make them susceptible to unexpected failures and efficiency issues [33], [42], [111]. To achieve effective monitoring, modern software systems generate large volumes of runtime metrics, such as response times or failure rates, which are used by dedicated teams to closely monitor runtime behavior and proactively detect possible issues [200]. These metrics are typically aggregated as time series data [127], and stored in specialized databases, such as Prometheus [128].

Recent advances in Time Series Forecasting (TSF) provide significant opportunities to enhance monitoring and quality assurance of software systems. TSF algorithms enable the prediction of future values in a time series by modeling patterns from historical data. Consequently, researchers have begun employing these algorithms to predict the future runtime behavior of software systems. Over the past few decades, TSF has been applied to support a variety of quality assurance tasks, including anomaly detection [5], [52], [55], software rejuvenation [53], capacity planning [74], [75], reliability prediction [129], [130], and self-adaptation [29], [56]. Therefore, understanding the most promising TSF algorithms for predicting future runtime metrics can yield broad improvements across numerous quality assurance activities, and it can guide the decisions of practitioners and researchers seeking to embed TSF in their quality assurance processes.

While software runtime metrics are notoriously difficult to analyze because of their inherent instability [2], [40], [57], [95], [131], TSF algorithms have demonstrated notable accuracy across various challenging domains, such as climate sciences [201] and epidemiology [202]. However, there is still little knowledge about how these methods specifically perform on software runtime metrics. To the best of our knowledge, the only study that empirically compares different TSF models on software runtime metrics is by Syu *et al.* [54], who examined TSF methods applied to web service quality attributes. However, that study suffers from several limitations: (i) it does not include more recent TSF models, such as recurrent neural networks, (ii) the evaluation is based on only two time series focused on a single runtime aspect (*i.e.*, response time), and (iii) the time series data were obtained from laboratory experiments rather than real-world software environments.

This chapter aims to bridge the knowledge gap by studying the effectiveness of Time Series Forecasting (TSF) methods applied to run-time software metrics, under

varying metric characteristics and temporal prediction demands.

To assess the suitability of TSF methods for predicting diverse runtime software metrics, we initially carried out a preliminary investigation [4], presented in Section 4.2, by conducting an empirical study on four TSF models applied to a small number of software metrics collected from a real-world IT infrastructure, covering three different software runtime aspects: *crash rates*, *hang time percentage* and *waiting time percentage*. Our evaluation compares one classical statistical forecasting model (*SARIMA*) and three recurrent neural network models with respect to two straightforward baselines, namely *seasonal naïve* (sNaïve) and *seasonal monthly mean* (sMM). With our preliminary investigation, we found out that TSF methods generally are more effective than the baselines for forecasting software runtime metrics, highlighting some interesting relationships among specific TSF methods and software metrics classes. Such findings motivated a more comprehensive investigation, encompassing additional forecasting approaches, expanded software metric categories, and a wider range of monitored applications. In that, Section 4.3 presents a substantial expansion of our prior research [4]. Specifically, we introduce the following extensions: (i) we increase the number of analyzed time series (by a factor of four), (ii) we include four additional runtime aspects, (iii) we evaluate twice the number of TSF models considered in our previous study, and (iv) we introduce a new class of TSF models: time series foundation models [132], [133], [203].

4.1 Background

4.1.1 Time Series Forecasting

Over the years, a vast variety of phenomena have been captured and modeled by leveraging the concept of time series, which can be defined as an ordered sequence of data points gathered at regular time intervals. Time Series Forecasting (TSF) approaches aim at predicting future evolution of the variable of interest by learning from its historical data while looking for recurring patterns in the data. Specifically, *trends*, *seasonality*, and *autocorrelation* concepts are widely used for modeling the stochastic process underlying the time series values. The "trend" describes the long-term underlying movement of the series, independent of the short-term fluctuations. The presence of a trend implies systematic and deterministic changes in the mean value of the series over time; in this case, the time series can be labeled as *non-stationary*. Vice-versa, when dealing with time series where the probabilistic behavior of every collection of values is identical to that of the time shifted set, we talk about *stationary* time series. This is a relevant concept since several forecasting approaches assume stationarity of the data or explicitly model trends in different ways. The "seasonality" concept arises when dealing with time series exhibiting patterns at a regular interval. Possible factors that may affect seasonality are holidays, business cycles, and regularly recurring workloads. When dealing with seasonal time series, using approaches capable of distinguishing the processes generating long-term values from those within the seasonal period can lead to better forecasting performance. Another fundamental concept used in this context is the "autocorrelation" function (ACF), measuring the linear predictability of the series with respect to its lagged version over successive time periods. The concept is similar to computing the correlation between two variables, but it is instead applied to two different versions of the same time series. Another related concept is the partial autocorrelation (PACF) that also accounts for all shorter lags together with the lag specified for the computation. Revealing such relationships among the data points within the time

series is an important insight for building an effective forecasting approach. Due to the easy porting of the time series concept, the effectiveness of TSF methods has been widely investigated in diverse areas, such as medicine, environment, system engineering, finance and more [129], [204], [205], [206], [207], [208], [209], [210].

TSF methods can be categorized into three broad categories based on their underlying methodological paradigms: *classical statistical methods*, *neural networks*, and *pre-trained transformer models*.

Classical Statistical Methods

Classical statistical TSF methods rely on well-established parametric approaches that model trend, seasonality, and autocorrelation through mathematical formulations. These methods typically assume stationarity or near-stationarity, making them suitable for cases where domain-specific insights can drive the modeling decisions. For instance, Autoregressive (AR) models capture temporal dependencies by expressing the current value of a time series as a function of its past observations. In contrast, Moving Average (MA) models account for short-term fluctuations by modeling the current value as a function of past error terms, effectively addressing irregular noise. A popular model in the literature is the ARIMA model, that combines these two components (*i.e.,* AR and MA) with an integration step to handle non-stationarity [211]. When dealing with data exhibiting regular and pronounced seasonality, SARIMA extends ARIMA by explicitly incorporating seasonal autoregressive and moving average components, enabling the modeling of recurring patterns that manifest at fixed seasonal intervals. ARIMA and SARIMA models are widely employed in time series forecasting studies in a variety of different domains [205], [206], [207]. Other methods in this category include Exponential Smoothing [212]—which, depending on the variant, can model trend, autocorrelation, and seasonality—and Prophet, [213] which includes non-linear trend components in addition to multiple seasonal frequencies.

Neural Networks

Neural networks have emerged as a viable alternative to classical statistical TSF methods [214], driven by the increasing availability of time series data. These methods follow a data-driven paradigm, leveraging deep learning to capture nonlinear and long-range dependencies within time series datasets [215], [216]. The typical process involves feeding a neural network with a time series by creating consecutive, shifted input windows to predict the datapoint(s) that follow the input sequence, thus framing the task as a supervised learning problem. Notably, neural networks have often outperformed classical statistical methods, particularly in scenarios characterized by intricate temporal patterns. Among the commonly employed neural network architectures for TSF are recurrent neural networks (RNNs), such as Fully-Connected Recurrent Neural Networks (FC-RNN) [217], [218], Gated Recurrent Units (GRU) [219], and Long Short-Term Memory (LSTM) networks [207], [220], [221]. FC-RNN is a neural network architecture in which the outputs are fed back into the network as inputs. Theoretically, this recurrent connection allows the network to maintain a state for holding information about previous inputs. The term "*fully connected*" denotes that each neuron in a given layer is connected to every neuron in both the preceding and subsequent layers. LSTM is a specialized form of RNN designed to address the challenge of learning long-term dependencies. Unlike standard RNNs, LSTMs incorporate gated cell states to adjust the flow of information.

These gates control the persistence and updating of information within the cell state. This architecture theoretically allows LSTMs to effectively retain important information over extended sequences while discarding irrelevant data, thus making them particularly suitable for TSF. GRU is another form of RNN, introduced to address the vanishing gradient problem associated with traditional RNNs. GRUs simplify the LSTM approach by combining the forget and input gates into a single "*update gate*" [219], while also merging the cell state and hidden state. This results in a more streamlined model that requires fewer parameters without sacrificing performance.

Time Series Foundation Models

Motivated by recent advances in pre-trained transformer models for natural language processing [222], [223], [224], researchers have begun developing time series foundation models [132], [133], [225], which are pre-trained on large time-series corpora comprising both real-world and synthetic datasets. The series used for training foundation models cover a wide range of domains, such as web search trends, energy, and healthcare, as well as datasets used in well-established forecasting competitions [132], [133]. Early studies suggest that these models can achieve competitive or even state-of-the-art performance across various TSF benchmarks, including zero-shot settings (*i.e.,* without prior learning on the target time series). This sharply contrasts with conventional neural network methods, which must be trained from scratch for each new time series. Their versatility makes them an appealing plug-and-play solution for practitioners. Each foundation model is explicitly designed with a specific architecture. Notable examples in this category include TimesFM [133], an encoder-only model, Chronos **ChronosFatir2024**, an encoder-decoder model based on the T5 architecture [226], and TimeGPT [225], another encoder-decoder model.

4.1.2 Time Series Forecasting in Software Engineering

TSF methods have been applied to a broad range of software engineering tasks. Seminal studies in this field have employed TSF methods to model the growth of software systems during evolution [227], [228]. More recently, Krishna *et al.* [135] used TSF to predict the number of bug reports and enhancement requests by analyzing historical issue reports.

Another line of recent research has leveraged TSF for anomaly detection in runtime software systems. For instance, Hundman *et al.* [52] employed LSTM to detect anomalies in spacecraft telemetry data. Similarly, Zhao *et al.* [55] used LSTM to predict the performance of software systems at run-time. RADig-X [5] utilized GRU and LSTM to detect anomalies in software crash trends within a large-scale IT infrastructure.

Researchers have also investigated the use of TSF for enabling proactive autonomous decisions in self-aware systems. For instance, Bauer *et al.* [29] conducted a preliminary study on integrating TSF methods into self-aware systems. Messias *et al.* [56] employed TSF to predict future workloads in cloud environments, leveraging genetic algorithms to combine multiple models. Other studies on workload prediction have also been proposed in [74], [75].

Additionally, TSF methods have been applied to predict the reliability of software systems. Amin *et al.* [129] used ARIMA models to forecast future software reliability, while Huitian *et al.* [130] applied exponential smoothing techniques for

a similar purpose. Jia *et al.* [53] applied a GRU model to predict memory resource consumption and determine the optimal rejuvenation time for cloud services.

To the best of our knowledge, the only empirical study that explicitly compares the effectiveness of various TSF methods in a software context was conducted by Syu *et al.* [54]. Their study evaluated the capabilities of different TSF methods for assessing web service quality attributes using runtime metrics obtained from laboratory experiments. Our work represents a significant advancement over this work, by incorporating many recent TSF methods, such as RNNs and time series foundation models, and applying them to runtime metrics gathered from real-world software applications rather than controlled laboratory settings.

4.2 Preliminary Study

In order to investigate the effectiveness of TSF methods to predict runtime software metrics, we initially conducted a preliminary study aiming to assess the suitability of a limited set of widely employed TSF methods when applied to predict software runtime metrics. In that, we pose the following question:

RQ$_{3.0}$ *To what extent are widely adopted TSF methods suitable for predicting runtime software metrics across different metric types and forecasting horizons?*

To answer this RQ, we conduct three different analyses. In particular, we analyze (i) the effectiveness of such TSF methods when applied to predict runtime software metrics (A_1), (ii) their forecasting accuracy over different classes of metrics (A_2), and (iii) the impact of the forecasting horizon on the methods' accuracy (A_3). We investigate the forecasting accuracy of four TSF methods, including a seasonal autoregressive moving average model (SARIMA) and three different recurrent neural networks, namely: fully-connected recurrent neural networks (FC-RNN), long-short memory networks (LSTM), and gated recurrent unit networks (GRU). The evaluation is performed on a range of diverse real-world runtime software metrics that are categorized into 3 classes (*i.e.*, *crash rate*, *hang time*, and *waiting time*) and gathered from 8 different software applications, resulting in a total of 14,575 individual data points. These metrics were recorded over a period of one and a half years on the large-scale IT infrastructure of a company[1], which comprises thousands digital devices. In this study, we distinguish between *short-* and *long-term* software metrics. This distinction refers to the distance of the forecasting target with respect to the current week of the software metric under analysis.

We show that, overall, RNN models are more effective than SARIMA and naive baselines, with FC-RNN providing the best forecasting accuracy. Nonetheless, our evaluation underscores the lack of a "silver bullet" method that outperforms all others across the considered metrics. For instance, we found that when dealing with specific classes of metrics, such as application waiting times, SARIMA offers the most accurate prediction. Furthermore, we found that benefits of using RNN models are valuable until the forecasting horizon does not exceed approximately one week.

This preliminary study provides (i) a first empirical assessment of TSF methods when applied to runtime software metrics, (ii) an investigation of how different TSF methods behave when applied to different classes of metrics, and (iii) a sensitivity analysis of TSF on runtime software metrics while varying forecasting horizons.

[1]Due to privacy concerns, the company choose not to reveal itself.

4.2.1 Design

In this section, we describe our experiment design, including the dataset, the TSF methods (and baselines) we studied, and the specific experimental procedures we used to gather metric predictions.

Dataset

Our dataset consists of 25 distinct runtime software metrics collected from the IT infrastructure of a large company, which includes more than 30k digital devices with heterogeneous hardware and software configurations. For each of the 25 metrics, the dataset provides a time series of daily observations collected over a period of one and a half years, thus resulting in a total of 583 measurements per metric. Each metric (*i.e.,* times series) concerns to a specific runtime aspect of a particular software application. For example, one metric might represent the *crash rate* of a specific web browser, while another could indicate the *hang time* of a particular email client. The dataset encompasses 8 distinct software applications from a diverse array of types, including web browsers, communication platforms, and word processors. Each application was monitored on 3 different runtime aspects, which we refer to as *metric classes*. In the following, we describe the semantics of each metric class.

- *Crash rate* denotes the average number of observed crashes of an application per hour of use. It is calculated by dividing the total number of observed application crashes within the IT infrastructure by the cumulative hours of application usage across all devices.
- *Hang time percentage* represents the percentage of application usage time during which the application remains in a "hang" state, *i.e.,* when the application is unresponsive and not performing any active processing tasks. This runtime aspect can be critical for diagnosing potential issues within software applications.
- *Waiting time percentage* is defined as the time users spend in waiting for an application to respond while it is actively running. This metric encompasses periods when the application is unresponsive, known as the "hang" state, as well as application and network loading times. It reports the percentage of the total application usage time that is spent in a "waiting" state across all devices within the IT infrastructure.

As a result, the dataset contains 24 time series (*i.e.,* 3 for each of the 8 software applications), plus an additional one that reports the *crash rate* related to operating system failures, such as *blue screen of death* [229] or *kernel panics* [230].

Due to privacy concerns, we cannot make publicly available the dataset used in our study.

Models and Baselines selection

For our empirical study, we selected four TSF methods, including one autoregressive moving-average model and three recurrent neural network models. We chose these types of models because they have been successfully applied to time series analysis in previous research [53], [129], [205], [220]. Specifically, our selection comprises the following TSF methods: (i) Seasonal Autoregressive Integrated Moving Average (SARIMA) [231], (ii) Fully Connected Recurrent Neural Network (FC-RNN) [217],

[218], (iii) *Long Short-Term Memory* (LSTM) [221] and (iv) *Gated Recurrent Unit* (GRU) [219].

SARIMA is an extension of the autoregressive integrated moving-average (ARIMA) model [211] that addresses the seasonal fluctuations often observed in time series data. It is based on the integration of additional seasonal terms, which allow the model to capture both short- and long-term dependencies, as well as repetitive patterns that occur at fixed intervals. This versatility makes it suitable for forecasting time series where the data exhibit periodicity.

FC-RNN is a variant of neural networks where the outputs are fed back into the network as inputs, thus forming directed cycles. This creates a recurrent connection pattern that allows the network to maintain a state that can theoretically hold information about previous inputs indefinitely. The term "fully connected" denotes that each neuron in a given layer is connected to every neuron in both the preceding and subsequent layers.

LSTM is a specialized form of RNN designed to address the challenge of learning long-term dependencies. Unlike standard RNNs, LSTMs include a series of gated cell states that regulate the flow of information. These gates control the persistence and updating of information within the cell state. This architecture allows LSTMs to effectively retain important information over extended sequences while discarding irrelevant data, thus making them particularly suitable for TSF.

GRU is a particular form of RNN, introduced to solve the vanishing gradient problem inherent to traditional RNNs. GRUs simplify the LSTM approach by combining the forget and input gates into a single "update gate" [219], while also merging the cell state and hidden state. This results in a more streamlined model that requires fewer parameters without sacrificing performance.

To aid the interpretability of results, we compare the forecasting accuracy of TSF models with the ones of two naïve baseline models, namely *seasonal naïve* (sNaïve) and *seasonal monthly mean* (sMM).

sNaïve operates on the assumption of temporal recurrence, by repeating the last observed values for future forecasts. In other words, this method projects the most recent seasonal data forward to the next equivalent season, thus assuming cyclical repetition. For instance, when predicting values for the upcoming week, the method simply replicates the observations from the previous week. This method is commonly used as baseline for comparative analysis in TSF studies [54], [205], [214], [232], [233], [234].

sMM leverages recent seasonal weekly trends to forecast future values. Specifically, it computes predictions by calculating the mean of data points that correspond to the same weekday in the preceding month. For example, to forecast the value for an upcoming Monday, sMM would average the values from all Mondays in the last month. Domain experts from the company that provided the dataset have recommended this baseline approach, as it reflects the established analytical practices of their software monitoring team.

Experimental procedure

To account for the distinct characteristics of the TSF methods investigated in our study, we adopt two distinct procedures tailored specifically for the RNN and SARIMA models, respectively.

SARIMA evaluation. For SARIMA, we independently create one model instance for each of the 25 time series (*i.e.,* runtime software metrics). The fitting process in SARIMA models consists of estimating the $(p, d, q)(P, D, Q)_s$ parameters, where: p and P represent the order of the autoregressive (AR) terms for the non-seasonal and seasonal parts respectively; d and D denotes the degree of differencing needed to render the series stationary on both non-seasonal and seasonal levels; q and Q indicate the order of the moving average (MA) terms for the non-seasonal and seasonal components; and s denotes the seasonality period of the time series data. To estimate the d and D parameter, we considered the number of times we applied differencing to obtain a stationary time series according to the ADFuller [235] test. The best values of (p, q, P, Q) have been searched by fitting the model with all the values ranging from 0 to 3 for each parameter, and by selecting the parameters configuration that gave the lowest *Akaike Information Criterion* (AIC) [236], [237] value after the model fitting. We set the s parameter to 7, which corresponds to a one week seasonality. The fitting process is performed using the initial 467 consecutive measurements of the time series (*i.e.,* 80% of the data points), with the remaining 20% of the measurements reserved for assessing the forecasting accuracy of the model. For the evaluation, starting from the initial input window of 467 consecutive measurements, we progressively increase the input window by one time unit and use the model to forecast the subsequent 14 measurements. This methodology enables us to simulate a realistic scenario of progressive forecasting, in which the model is continuously tested as new data becomes available in the time series. As a result, we obtain for each time series a set of 89 *forecast segments* F, each consisting of 14 predicted measurements.

RNN evaluation. To train the RNN models, instead, we employ a *sliding window segmentation* technique [238], which divides a longer time series of measurements into smaller, overlapping segments of fixed size. Specifically, we use a sliding window of 28 consecutive measurements (*i.e.,* 4 weeks) to generate multiple overlapping segments. The initial 14 measurements serves as the input segment (or *look-back* period), whereas the remaining measurements form the *forecast segment*. Following the best practice for deep-learning models [208], [239], we scaled the data using the *z-score* normalization technique, thus ensuring that the data falls within a comparable range. Specifically, we standardize each of the 28 measurements within the window segment by using the process outlined in Equation (4.1):

$$z_i = \frac{x_i - \mu_{input}}{\sigma_{input}} \tag{4.1}$$

where z_i is the standardized value of the i^{th} data point of the window segment, x_i is the original data point value, μ_{input} is the mean of the input segment measurements, and σ_{input} is their standard deviation. We use the mean and standard deviation of the input segment, instead of the entire window of 28 measurements, to ensure that our evaluation process reflects realistic forecasting scenarios, where future measurements (*i.e.,* the forecast segments) are not yet known.

To evaluate a RNN model, we independently train one model instance for each of the 25 time series, by using the initial 72% consecutive window segments (*i.e.,* 394 segments) for training, the subsequent 8% for validation (*i.e.,* 19 segments), and the remaining 20% for testing (*i.e.,* 89 segments). For the RNN models under consideration (*i.e.,* FC-RNN, LSTM, GRU), the neural network architecture is designed as follows. The first layer of the architecture consists of 14 units of the specific RNN type

(either FC-RNN, LSTM, or GRU). The architecture includes a second layer composed of another 14 units of the same type, which is encapsulated within a bidirectional layer. Finally, the processed information is channeled through a fully connected layer with 14 units, by employing the hyperbolic tangent (tanh) activation function to produce the final output. We use the Adam optimizer [192] with an initial learning rate of 0.001, using the mean absolute error (MAE) as the loss function. Additionally, we implement a reduce-on-plateau strategy, which decreases the learning rate when no improvement is observed in the validation loss for 20 epochs. The training is conducted for a maximum of 500 epochs, with an early stopping mechanism that terminates the process if no improvement in the validation loss is observed for 50 consecutive epochs. The best model is selected basing on the lowest validation loss observed throughout the training epochs. As a result of the evaluation process, similarly to SARIMA, we obtain for each RNN model and time series a set of forecast segments F, each consisting of 14 measurements.

Notation. For notational convenience, we use F^i to denote the forecast segment that aims to predict the time series data points beginning from the i^{th} position. For instance, F^{500} represents the forecast segment predicting data from the 500^{th} to the 513^{th} position of the time series. Additionally, we use F_j to denote the j^{th} element of the forecast segment, with $1 \leq j \leq 14$. Figure 4.10 graphically illustrates the example presented above (F^{500}). The black circles represent the data used as input by the models and the red ones are the predicted measurements.

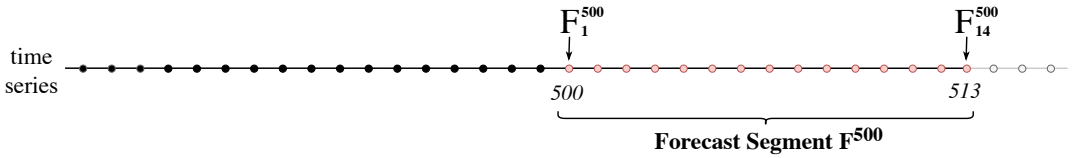


FIGURE 4.1: Forecast segment within the time series.

Ultimately, for each TSF method and time series, we obtain 89 forecast segments F^i, with $482 \leq i \leq 570$, where 482 denotes the starting position of the first forecast segment appearing in the time series, and 570 denotes the starting position of the last forecast segment of the time series.

Implementation details. The experiment has been implemented in *Python*. The `SARIMAX` class of *statsmodels*[2] library has been used for implementing SARIMA models. RNNs have been implemented using `SimpleRNN` (for FC-RNN models), `LSTM` and `GRU` classes of *Tensorflow*[3].

4.2.2 Preliminary Results

In this section, we discuss the preliminary results along the three analyses presented in Section 4.2.1. For sake of clarity, we present each analysis in a separate subsection. Each subsection is structured as follows: we first outline the objective of the analysis, then we describe the methodology and discuss the results.

[2] https://www.statsmodels.org/stable/index.html
[3] https://www.tensorflow.org/

A_1: On the effectiveness of TSF methods applied to predict runtime software metrics.

Objective. Through this first analysis, we investigate the effectiveness of TSF methods in predicting runtime software metrics. Specifically, our focus is on evaluating the accuracy of these methods in "*one-step-ahead*" forecasting, which entails predicting runtime software metrics for the immediate next day. We adopt this setting because it is commonly used in TSF studies [54], [232], [240], [241].

Methodology. We employ a commonly used metric for measuring forecasting accuracy, namely SMAPE (Symmetric Mean Absolute Percentage Error).

The values of SMAPE range from 0% to 200%, where 0% indicates perfect forecasting accuracy, while 200% represents the worst accuracy. Given that the aim of this analysis is to evaluate the forecasting accuracy of TSF methods in predicting next-day metrics, we calculate the SMAPE by focusing exclusively on the first element of each forecast segment F^i, *i.e.*, the next-day prediction.

Formally, given a time series Y and a particular TSF method, we calculate the corresponding SMAPE as follows:

$$\text{SMAPE} = \frac{100\%}{n-k+1} \sum_{t=k}^{n} \frac{\left|F_1^t - Y_t\right|}{\frac{1}{2}\left(\left|F_1^t\right| + \left|Y_t\right|\right)} \tag{4.2}$$

where k and n denote the positions of the first and last elements of the time series used for evaluation (specifically, k=482 and n=513 in our case). F_1^t denotes the first element of the forecast segment F^t (*i.e.*, the next-day prediction of the TSF method for the t^{th} element of the time series), and Y_t denotes the actual time series measurement at position t.

As a result of this process, for each TSF method, we obtain 25 SMAPE results, *i.e.*, one per time series.

In order to assess (and compare) the forecasting accuracy of different TSF methods, we plot the SMAPE distribution of each TSF method using box plots, and report the associated descriptive statistics (*e.g.*, mean, median). Additionally, we conduct an analysis to determine whether each TSF method outperforms the naive baselines. Indeed, the practical applicability of a TSF method may be questioned if it does not improve upon these baselines. To accomplish this, we employ the Wilcoxon signed-rank test [242] for each pair of <*TSF method, baseline*> to compare their respective SMAPE values. We set the significance level at 0.05, meaning that differences with p-values below this threshold are considered statistically significant. In addition to the Wilcoxon signed-rank test, we utilize the common language effect size [243], in the version proposed by Vargha and Delaney ($\hat{A}_{12}$) [158], to assess the magnitude of the differences observed. Given two related paired samples X and Y, the common language effect size is the proportion of pairs where X is higher than Y.

$$\hat{A}_{12} = P(X > Y) + .5 \times P(X = Y) \tag{4.3}$$

The $\hat{A}_{12}$ value, which ranges from 0 to 1, is interpreted using the thresholds provided by Vargha and Delaney [158]. A $\hat{A}_{12}$ value of 0.5 suggests that there is no significant difference in forecasting accuracy between the TSF method and the baseline. A value of $\hat{A}_{12}$ larger than 0.5 indicates that the TSF method is likely to yield more accurate forecasts than the baseline, *i.e.*, lower SMAPE. Specifically, the magnitude of the difference is considered as *small* (S^+), *medium* (M^+) and *large* (L^+) if the $\hat{A}_{12}$ value is greater than or equal to 0.56, 0.64 and 0.71, respectively. Conversely, a

value of $\hat{A}_{12}$ lower than 0.5 indicates that the TSF method yields worse forecasting accuracy than the baseline. In this case, the effect size is considered as *small* (S^{-}), *medium* (M^{-}) and *large* (L^{-}) if the $\hat{A}_{12}$ value is lower than or equal to 0.44, 0.34 and 0.29, respectively. We consider comparisons that report $\hat{A}_{12}$ larger than 0.44 and lower than 0.56 as negligible, and therefore not meaningfully different.

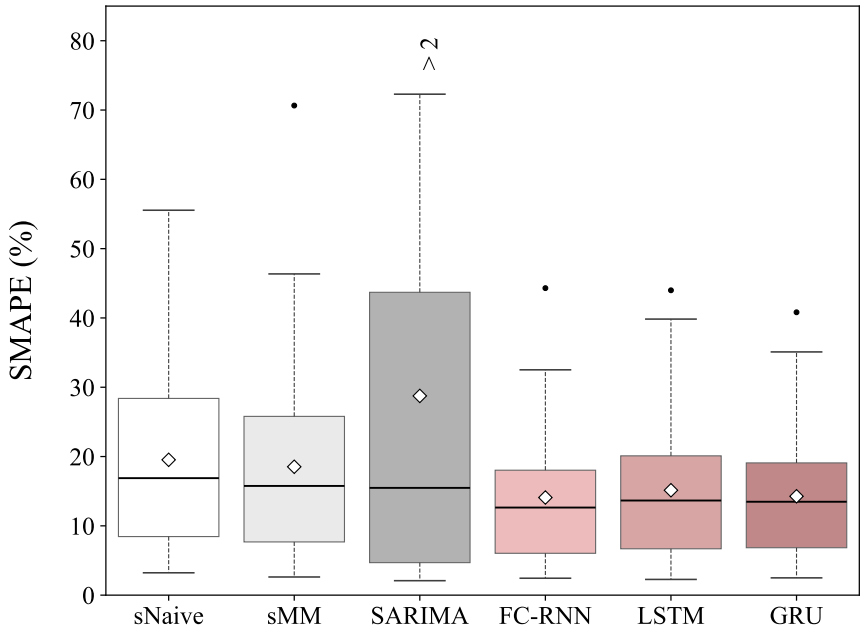


FIGURE 4.2: A_1. SMAPE distribution for each TSF Method. Boxplots are highlighted in red for RNN models, grey for SARIMA and white/silver for the baselines.

	SMAPE Statistics				
	Mean (μ)	Median ($\tilde{x}$)	Std Dev (σ)	Min	Max
sNaïve	19.53	16.88	14.23	3.23	55.54
sMM	18.52	15.76	15.77	2.62	70.65
SARIMA	28.75	15.48	33.02	2.09	116.95
FC-RNN	14.09	12.64	10.31	2.44	44.30
LSTM	15.14	13.65	10.94	2.27	43.99
GRU	14.26	13.48	9.91	2.49	40.82

TABLE 4.1: A_1. SMAPE descriptive statistics for TSF methods and baselines.

Results. We reported the results concerning the SMAPE distribution of TSF methods in Figure 4.2 and Table 4.1. A first observation is that RNN-based methods demonstrate promising forecasting accuracies, as evidenced by their average SMAPE values: 14.09% for FC-RNN, 15.14% for LSTM, and 14.26% for GRU. Among them, FC-RNN appears to be the most effective, by providing the lowest median and mean SMAPE values. These results are comparable to, or even better than, those reported in recent TSF research [29], [232], [244]. From Figure 4.2, it can be observed that

	sNaïve	sMM
SARIMA	0.50 (-), p=0.596	0.48 (-), p=0.164
FC-RNN	**0.63 (S^+), p=<0.001**	**0.57 (S^+), p=0.001**
LSTM	**0.59 (S^+), p=<0.001**	0.55 (-), p=0.027
GRU	**0.60 (S^+), p=<0.001**	**0.56 (S^+), p=0.006**

TABLE 4.2: A_1. Results of the comparison between TSF methods and baselines. Each cell is formatted as "$\hat{A}_{12}$, *p-value*", where $\hat{A}_{12}$ represents the Vargha-Delaney effect size, and the p-value is the result of the Wilcoxon signed-rank test. The interpretation of the $\hat{A}_{12}$ value is also provided in brackets. Comparisons where TSF methods outperform baselines with statistical significance (*p-value*$<$ 0.05) and a non-negligible effect size are highlighted in bold.

RNN-based methods (shown in red in Figure 4.2) demonstrate better forecasting accuracy than baselines, as indicated by their lower SMAPE values. In support of this, we also notice in Table 4.1 that RNN models show considerably lower mean and median SMAPE values than those provided by baselines. For example, if we compare the worst-performing RNN model in terms of mean and median SMAPE (*i.e.,* LSTM) with the best-performing baseline (namely, sMM) we still observe an improvement in mean (18.52% versus 15.14%) and median (15.76% versus 13.62%) values. Another interesting result is that RNN-based methods exhibit higher stability in forecasting accuracy when compared to other approaches, *i.e.,* the prediction error tends to vary less from one time series to another. Indeed, as shown in Table 4.1, RNN-based methods exhibit lower standard deviations in SMAPE. Specifically, among RNN-based methods we observe a maximum standard deviation of 10.94% (LSTM), which is notably lower than the 14.23% and 15.77% standard deviations observed for sNa"ive and sMM baselines, respectively. This observation is remarked by Figure 4.2, which displays a narrower inter-quartile range (IQR) for RNN-based methods that is also more shifted towards the bottom, thus indicating lower errors. This suggests that RNN-based methods provides more accurate and stable prediction than baselines. Further confirmation of this finding comes from the results of the Wilcoxon signed-rank test presented in Table 4.2. Both FC-RNN and GRU demonstrate statistically significant improvements over the baselines ($p < 0.05$), with a non-negligible effect size ($\hat{A}_{12} \geq 0.56$). Additionally, LSTM outperforms sNaive with a small effect size. These results indicate considerable benefits in employing RNN for TSF of runtime software metrics.

An analysis of Figure 4.2 reveals that SARIMA results in the widest SMAPE IQR, thus indicating significant variation in its forecasting accuracy. This observation is further supported by the data in Table 4.1, which shows SARIMA having the highest standard deviation, at 33.02%, among all evaluated methods. Moreover, SARIMA provides the worst average forecasting accuracy, with an average SMAPE of 28.75%. While these results might suggest that SARIMA is poorly suited for predicting runtime software metrics, a closer examination of its SMAPE distribution reveals some interesting insights. For example, as shown in Table 4.1, SARIMA achieves the lowest minimum SMAPE value (2.09%) and its median SMAPE (15.48%) is lower than those of the naive baselines. Additionally, Figure 4.2 shows that SARIMA provides the lowest first quartile in SMAPE. These observations indicate that SARIMA performs quite well on a specific subset of time series, potentially even outperforming other approaches on these instances.

Summary. RNN-based methods are more effective in predicting runtime software metrics, with FC-RNN being the most effective one (average SMAPE of 12.64%). FC-RNN and GRU outperform both baselines with statistical significance ($p < 0.05$) with non-negligible effect sizes ($\hat{A}_{12} \geq 0.56$). SARIMA is less stable in forecasting accuracy, by exhibiting SMAPE values that significantly vary from one time series to another (standard deviation of 33.02%). To answer A_1, our analysis demonstrates that TSF methods, particularly RNN-based methods, are quite effective in accurately predicting runtime software metrics. These findings highlight the potential benefits of applying TSF methods into real-world software monitoring contexts.

A_2: Analysis of the forecasting accuracy over different classes of metrics.

Objective. The second analysis considers the forecasting accuracy of TSF models for each class of runtime software metric (*i.e., crashes rate/hang time perc./waiting time perc.*). We want to study whether TSF methods perform consistently across diverse metric classes or their accuracy varies significantly depending on the specific class of metric being forecasted.

Methodology. We reuse the SMAPE values previously calculated for A_1. However, rather than examining these values in aggregate, we group them per metric class. We investigate the forecasting accuracy of TSF methods across each metric class by analyzing their corresponding SMAPE distributions through box plots, and by examining the associated descriptive statistics. For each metric class, we also compare the forecasting accuracy of TSF methods with naive baselines using Wilcoxon signed-rank test. This is done for each <*TSF method, baseline*> pair to determine if there is a statistically significant difference in their SMAPE values. Additionally, we employ the Vargha-Delaney $\hat{A}_{12}$ to assess the effect size.

Results. The SMAPE distribution computed over all the metric classes is depicted in Figure 4.3, and the associated descriptive statistics are reported in Table 4.3. We examined the results in two ways: (i) by evaluating the general forecasting accuracy of TSF methods on individual software metric classes, and (ii) by analyzing how each TSF method behaves in relation to each software metric class.

With the first analysis, we want to study how TSF methods collectively perform when applied to different classes of metrics. In Figure 4.3, we observe that SMAPE box plots for same metric classes exhibit similar behavior across various TSF methods. For instance, when observing the SMAPE distribution related to *waiting time perc.*, we notice significantly lower errors compared to those reported for other metric classes. This can be observed in both Figure 4.3 and Table 4.3: the IQR, mean, and median of SMAPE for *waiting time perc.* are consistently lower than those reported for other metric classes, regardless of the TSF method employed. This suggests that time series pertaining to the same metric classes may share common characteristics that influence the forecasting accuracy of TSF methods, thus leading to consistently better or worse outcomes. Indeed, time series related to *waiting time perc.* appear significantly more predictable than those related to *crash rate* or *hang time perc.*. Both *hang time perc.* and *crash rate* show larger and more dispersed (*i.e.,* higher standard deviation σ) SMAPE values, with *crash rate* looking as the hardest to predict. A possible motivation for this result could be that *waiting time perc.* metrics exhibit a seasonal pattern that recurs over time, while *crash rate* (or *hang time perc.*) metrics are influenced by more unpredictable factors, such as unexpected software bugs or hardware malfunctions.

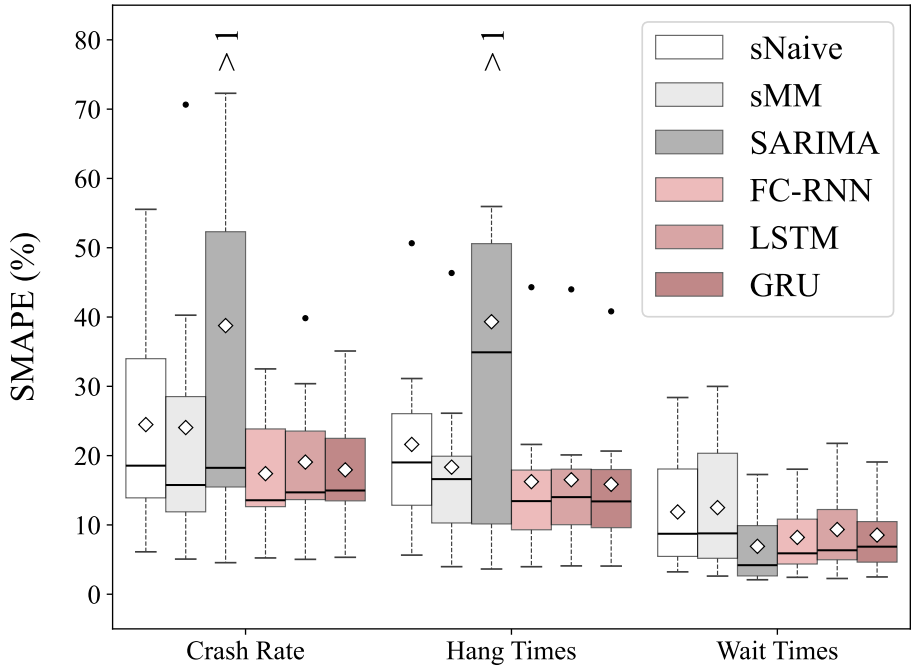


FIGURE 4.3: A_2. SMAPE distribution of TSF methods grouped by metric class.

		Baselines		TSF Methods			
		sNaive	sMM	SARIMA	FC-RNN	LSTM	GRU
Metric Class	SMAPE Stat						
Crash Rate	Mean (μ)	24.48	24.05	38.76	17.40	19.07	17.95
	Median ($\tilde{x}$)	18.55	15.76	18.24	13.56	14.70	14.97
	Std Dev (σ)	16.40	20.88	37.06	9.72	11.21	9.82
	Max	55.54	70.65	116.95	32.51	39.83	35.10
	Min	6.12	5.07	4.55	5.24	5.03	5.33
Hang Time Perc.	Mean (μ)	21.62	18.33	39.32	16.24	16.52	15.86
	Median ($\tilde{x}$)	19.02	16.61	34.91	13.44	14.01	13.40
	Std Dev (σ)	14.31	13.18	36.52	12.70	12.33	11.48
	Max	50.65	46.35	116.26	44.30	43.99	40.82
	Min	5.65	3.98	3.64	3.97	4.09	4.07
Waiting Time Perc.	Mean (μ)	11.87	12.50	6.91	8.20	9.34	8.53
	Median ($\tilde{x}$)	8.72	8.78	4.19	5.90	6.34	6.86
	Std Dev (σ)	8.85	9.98	5.77	6.01	7.38	6.03
	Max	28.39	29.99	17.28	18.04	21.77	19.08
	Min	3.23	2.62	2.09	2.44	2.27	2.49

TABLE 4.3: A_2. SMAPE descriptive statistics for TSF methods and baselines grouped by software metric class.

		Software Metric Class		
TSF Method	Baseline	Crash Rate	Hang Time Perc.	Waiting Time Perc.
SARIMA	sNaive	0.42 (-), p=0.129	0.36 (-), p=0.312	**0.73 (L^+), p=0.008**
	sMM	**0.38 (S^-), p=0.008**	0.34 (-), p=0.148	**0.70 (M^+), p=0.008**
FC-RNN	sNaive	**0.65 (M^+), p=0.004**	**0.64 (M^+), p=0.008**	**0.69 (M^+), p=0.008**
	sMM	0.52 (-), p=0.496	**0.59 (S^+), p=0.008**	**0.64 (M^+), p=0.016**
LSTM	sNaive	**0.60 (S^+), p=0.012**	**0.62 (S^+), p=0.008**	0.61 (-), p=0.109
	sMM	0.53 (-), p=0.57	0.56 (-), p=0.055	0.59 (-), p=0.109
GRU	sNaive	**0.59 (S^+), p=0.02**	**0.64 (M^+), p=0.008**	0.61 (-), p=0.055
	sMM	0.52 (-), p=0.652	**0.58 (S^+), p=0.023**	**0.62 (S^+), p=0.023**

TABLE 4.4: A_2. Results of the comparison between TSF methods and baselines over each class of metric. Each cell is formatted as "$\hat{A}_{12}$, *p-value*", where $\hat{A}_{12}$ represents the Vargha-Delaney effect size, and the p-value is the result of the Wilcoxon signed-rank test. The interpretation of the $\hat{A}_{12}$ value is also provided in brackets. Comparisons where TSF methods outperform baselines with statistical significance (*p-value*< 0.05) and a non-negligible effect size are highlighted in bold.

In our second analysis, we investigate the forecasting accuracy of various TSF methods for each class of runtime software metric. In Figure 4.3 we observe that RNN-based methods, *i.e.,* FC-RNN, LSTM, and GRU, perform generally well across the different metric classes. RNN-based methods show lower (or comparable) SMAPE distributions than those reported by naive baselines. This is confirmed by the results of the Wilcoxon signed-rank test, reported in Table 4.4. For instance, FC-RNN outperforms sNaive in all the metric classes with a statistically significant difference ($p < 0.05$) and a medium effect size ($\hat{A}_{12} \geq 0.64$). LSTM also outperforms sNaive with a small effect size ($\hat{A}_{12} \geq 0.56$) on *crash rate* and *hang time perc*. GRU, similarly, outperforms sNaive with small and medium effect sizes on *crash rate* and *hang time perc*, respectively. Compared to the sMM baseline, both LSTM and GRU report statistically significant lower SMAPE values on *hang time perc.* and *waiting time perc.*, with either small or medium effect sizes. Overall, FC-RNN demonstrates the best forecasting accuracy across the different metric classes, by providing statistically significant improvement over both naive baselines in all metric classes, except on *crash rate* when compared to sMM.

Another noteworthy result concerns the diverse forecasting accuracy provided by SARIMA over different metric classes. By examining Figure 4.3, we observe that SARIMA provides substantially different SMAPE values over different metric classes. For instance, while it reports relatively high errors for *crash rate* and *hang time perc.* (*e.g.,* average of 38.79% and 39.32%, respectively), considerably low SMAPE values are reported for *waiting time perc.* (*e.g.,* average of 6.91%). This diversity is remarked in the comparison with naive baselines. According to Table 4.4, SARIMA outperforms both sNaive and sMM on *waiting time perc.* with statistically significant difference, and large and medium effect sizes, respectively. However, for other metric classes, SARIMA does not provide any improvement over the baselines. Even more, it provides worse forecasting accuracy than sMM on *crash rate*. Nonetheless, by analyzing both Figure 4.3 and Table 4.3, we can notice that SARIMA provides the best forecasting accuracy on *waiting time perc.* among different TSF methods, with the lowest mean and median values (respectively, 6.91% and 4.19%). This may suggest that such method performs particularly well when dealing with more predictable time series that show recurring patterns, while it may not fit well on more irregular runtime software metrics, such as *crash rate* or *hang time perc*.

Summary. When collectively analyzed, TSF methods exhibit diverse forecasting accuracy depending on the class of runtime software metric. For instance, TSF methods show higher effectiveness when dealing with *waiting time perc.* metrics, while they are less effective on classes of metrics more closely related to software malfunctions, such as *hang times perc.* and *crash rate*. RNN-based methods demonstrate the most consistent effectiveness across different metric classes, with FC-RNN being the most effective one. However, when dealing with classes of more predictable metrics, such as *waiting time perc.*, SARIMA has been shown to be the most effective.

A_3: Impact of forecasting horizon on the predictions accuracy.

Objective. Our objective here is to assess how forecasting accuracy decreases when predicting longer-term runtime software metrics. The TSF methods that we consider are able to generate multi-step ahead forecasts. This means that, at any given time, a TSF method can be queried to predict (for instance) the runtime software metrics of the next 14 days. It is expected that near-term predictions (*e.g.*, for the forthcoming days) will generally be more accurate than those for longer terms (*e.g.*, the following week). Through this analysis, we aim to evaluate the extent to which the accuracy decreases as the forecasting horizon is progressively extended.

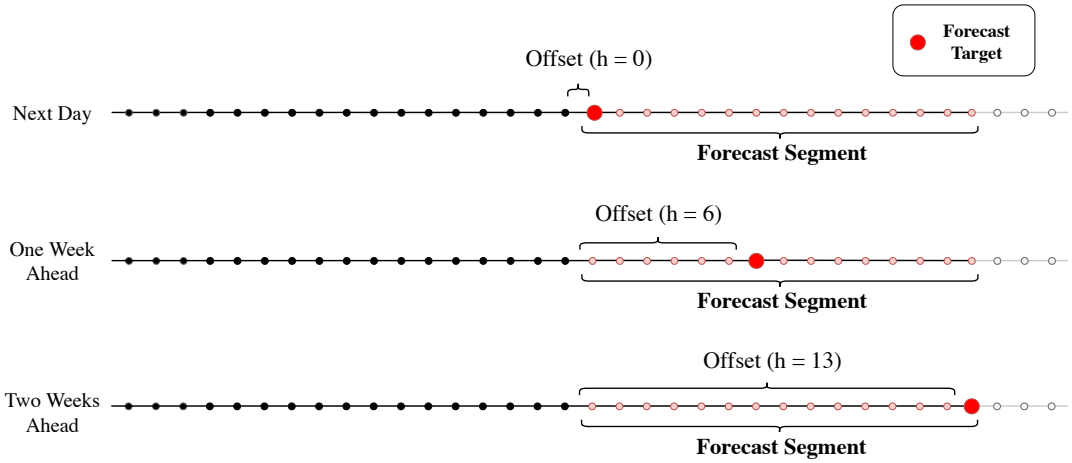


FIGURE 4.4: A_3. Three representative offset scenarios: next day ($h = 0$), one week ahead ($h = 6$), and two weeks ahead ($h = 13$).

Methodology. To achieve this research goal, we introduce the concept of *offset* (h), which we define as the number of days between the last time step used as input and the *forecast target*. This essentially simulates scenarios where, at a given moment, the goal is to predict the metric for a specific future day. For example, an offset $h = 0$ indicates a scenario where the forecast target is the next day, while an offset $h = 6$ corresponds to a scenario where the goal is to predict runtime software metrics for the same day in the following week. Figure 4.4 graphically illustrates the concept of offset and forecast target. The first scenario represents the case where the goal is to predict the metrics of next day ($h = 0$), the second scenario targets predictions for the same day in the following week ($h = 6$), and the third scenario uses the same day of two weeks ahead as forecast target ($h = 13$).

For each specific offset h, we calculate the SMAPE of a given TSF method applied to a particular time series Y as follows:

$$\text{SMAPE} = \frac{100\%}{n-k+1} \sum_{t=k}^{n} \frac{\left|F^t_{1+h} - Y_{t+h}\right|}{\frac{1}{2}\left(\left|F^t_{1+h}\right| + \left|Y_{t+h}\right|\right)} \tag{4.4}$$

where F^t_{1+h} represents the $(1+h)^{th}$ element of the forecast segment F^t, corresponding to the prediction for the $1+h$ steps ahead day (*i.e.*, the forecast target). Y_{t+h} indicates the actual measurement observed in the time series on that particular day. For example, with a $h = 6$ offset, SMAPE is calculated by considering exclusively the errors at the 7^{th} elements across all forecast segments of the time series. This corresponds to assessing the forecasting accuracy of a TSF method in predicting the runtime software metric for the same day in the following week.

As results of this process, for each TSF method, we compute 350 SMAPE values, corresponding to each combination of time series and offset h, with $0 \leq h \leq 13$.

We employ box plots and line plots to illustrate the relationship between SMAPE and the offset h. The analysis of these plots aids in gaining a better understanding of how forecasting accuracy degrades as the forecasting horizon is extended.

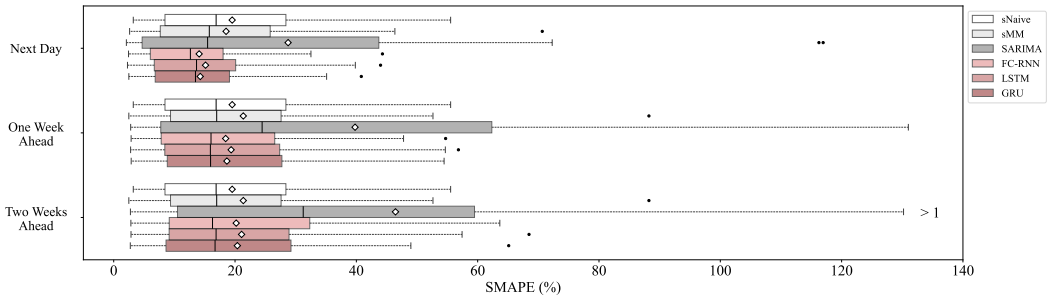
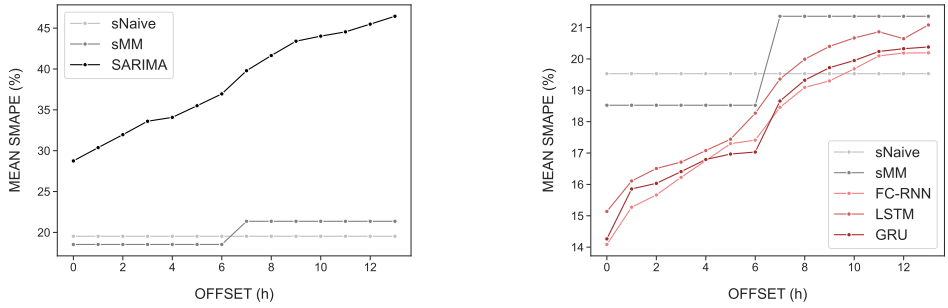


FIGURE 4.5: A_3. SMAPE distribution of TSF methods under three representative offset scenarios: next day ($h = 0$), one week ahead ($h = 6$), and two weeks ahead ($h = 13$).



(A) SARIMA vs baselines.

(B) RNNs vs baselines.

FIGURE 4.6: A_3. Relationship between mean SMAPE and offset (h).

Results. Figure 4.5 displays the SMAPE distribution of each TSF method for three representative offset scenarios: next day (h=0), one week ahead (h=6), and two weeks ahead (h=13). Figure 4.6 shows the trend of the mean SMAPE of each TSF method as the offset h increases. Figure 4.7 reports the same information for each metric class, separately.

Figure 4.6 clearly highlights an increasing trend in the mean SMAPE as the offset increases. This increasing trend is also observable in individual classes of metrics,

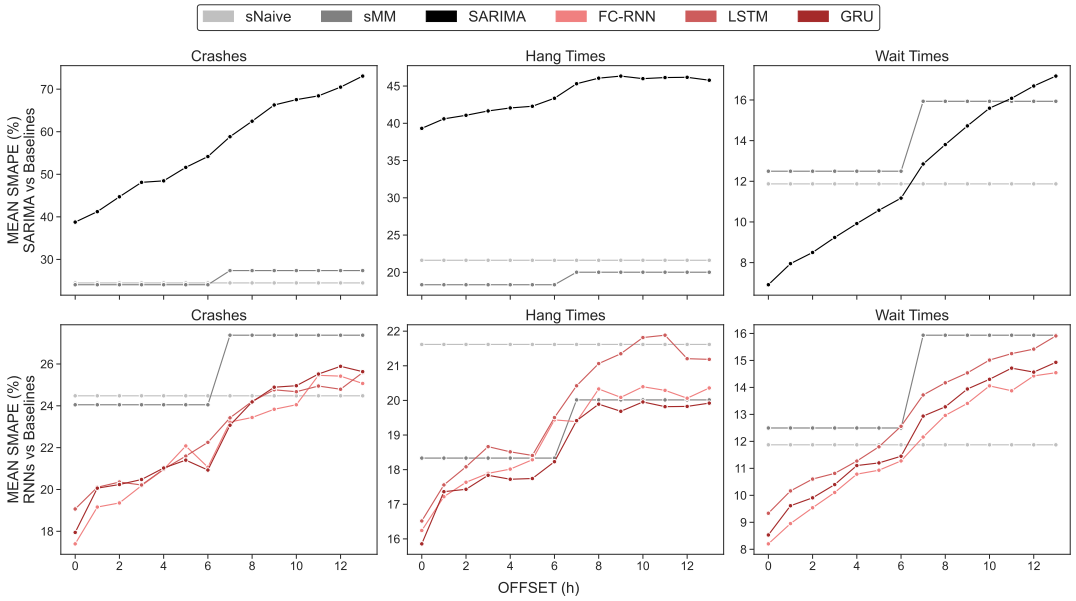


FIGURE 4.7: A_3. Relationship between mean SMAPE and offset (h) grouped by metric class. The first row displays results for SARIMA, while the second row shows the results of RNN-based methods.

as shown in Figure 4.7. By looking at Figure 4.5, we can also notice that the SMAPE tends to become more variable as the offset increases, thus indicating less stability in forecasting accuracy on long-term predictions. This is particularly visible in RNN-based methods, which show an IQR that consistently increases with each offset scenario.

Another interesting outcome of our analysis concerns the comparison of the forecasting accuracy of TSF methods with the baselines. Specifically, upon examining Figure 4.6, we notice a critical offset beyond which RNN-based methods begin to achieve comparable or even worse forecasting accuracy than the ones of baselines. This specific turning point is observed at approximately a week ahead ($h \approx 7$). A similar behavior is also observable on each individual class of runtime software metric, as shown in Figure 4.7. Nonetheless, each class of metric exhibits slightly different patterns. For instance, in Figure 4.7, we notice that mean SMAPE of RNNs begin to exceed those of baselines at offset $h = 9$ on *crashes rate*, instead RNNs start to exceed baselines error at offset $h = 6$ for *waiting time perc*. This finding is also evident in Figure 4.5, where a clear degradation is observed in the SMAPE distribution, moving from the *next day* to the *one week ahead* scenario. Overall, these results suggest that the benefits of employing RNNs for predicting runtime software metrics vanish when the prediction target exceeds the current week. Our results highlight the importance of accounting for the changing dynamics of runtime software metrics, thus outlining the necessity of incorporating recent temporal patterns. In a practical software monitoring context, this emphasizes the need to regularly update predictions as new data becomes available.

Another interesting observation is that, albeit RNN models exhibit similar behavior, in some cases they start to exceed the baselines at different offsets. For example, by looking at Fig. 4.7, we noticed that LSTM start to exceed the sMM on *hang time perc.* as early as offset is equal to $h = 3$, while for FC-RNN this happens at offset $h = 6$. Furthermore, on *hang time perc.*, GRU outperforms the baselines across the entire offset range, although from offset $h = 6$ its SMAPE becomes very similar to that

of sMM. This reveals that the choice of the specific RNN model can have non-trivial impact on the forecasting accuracy of longer-term runtime software metrics.

We then examined the accuracy of TSF models in detail for each class of software metrics. In line with the findings discussed in A_2, Figure 4.7 reveals that predictions for *waiting time perc.* metrics consistently displayed lower errors than other metric classes, even in longer-term forecasts, with a maximum mean SMAPE of 16%. It is interesting to note that SARIMA, which is the best-performing method for *waiting time perc.* according to A_2, shows here a similar behavior to that of RNN-based methods. In fact, we observe that SARIMA only begins to underperform the baselines when the offset exceeds one week ($h = 6$).

Summary. Concerning A_3, our findings reveal that short-term forecasts are consistently more accurate than longer-term ones. Namely, the SMAPE values demonstrate a rising trend as the offset increases. Our results suggest that the advantages of using TSF methods vanish if the offset exceeds approximately one week. These findings offer practical insights into the suitability of TSF methods for predicting long-term runtime software metrics.

4.2.3 Overview of Early Findings

Aiming to answer $RQ_{3.0}$, the preliminary study evidences that widely employed TSF methods globally outperform both sNaïve and sMM baselines when predicting runtime software metrics, underscoring their suitability in software monitoring scenarios. In addition, we observed that the advantages of forecasting approaches diminish beyond a specific prediction horizon, suggesting that it is worthwhile to carefully assess whether to use such complex methods for generating long-term predictions. These results offer preliminary evidence of the potential of TSF methods when applied to runtime software metrics, which are highly variable and influenced by external factors such as complex workload dynamics, and encourage future research to expand upon the depth and breadth of our study scope. Moreover, we noticed the absence of a single "silver bullet" model outperforming all the others in every scenario when forecasting runtime software metrics. Specifically, when forecasting the *waiting time perc.* series, the SARIMA model performs best, although it performs worse for the other classes. This variability in model performance across metric classes further motivates the need for a more comprehensive investigation to validate this behavior over a broader selection of models and runtime metric categories.

4.3 Advanced Study

The preliminary study presented in Section 4.2 has empirically shown that a limited set of four widely employed TSF methods significantly outperform straightforward baselines when forecasting runtime software metrics, supporting their suitability in this context. This advanced study extends the earlier analysis by expanding the pool of forecasting models and the dataset employed, aiming to provide a more comprehensive evaluation of TSF methods in this context. To this end, we pose the following research questions:

$RQ_{3.1}$ *Are the selected TSF methods suitable for predicting software runtime metrics?*

$RQ_{3.2}$ *Which TSF methods provide the most accurate predictions of software runtime metrics?*

RQ$_{3.3}$ *Do TSF methods exhibit diverse forecasting accuracy over different classes of software runtime metrics?*

RQ$_{3.4}$ *To what extent does TSF accuracy change when extending the forecasting horizon?*

To answer our research questions, we investigate the forecasting accuracy of eight TSF methods, including three classical statistical models (i.e., *SARIMA* [231], *Prophet* [213], and *Exponential Smoothing* [212]), three different recurrent neural networks (i.e., *fully-connected recurrent neural networks* [217], [218], *long short-term memory networks* [221], and *gated recurrent unit networks* [219]), and two time series foundation models (*i.e., Chronos* [132] and *TimesFM* [133]).

Our evaluation is conducted on 110 real-world software runtime metrics, categorized into seven classes (*e.g.,* crash rate, hang time), gathered from 29 different software applications, resulting in a total of 44,000 individual data points. These metrics were recorded over approximately one year within the large-scale IT infrastructure of a company comprising thousands of digital devices.

Our results show that time series foundation models are by far the most effective, with *Chronos* providing the highest forecasting accuracy. This finding highlights the potential of these models to transfer knowledge from other domains into the software runtime domain, making them an attractive solution for software engineers aiming to integrate TSF into their quality assurance processes in a zero-shot setting.

However, our evaluation also emphasizes the absence of a universal "silver bullet" method that consistently outperforms all others across the considered metrics. In other words, foundation models may not always deliver the best forecasting accuracy. In such cases, more careful model selection can sometimes lead to significant improvements in forecasting performance.

Our key contributions are as follows:

- We conduct a comprehensive empirical assessment of eight TSF methods on 110 time series covering seven different runtime software aspects.
- We investigate how these methods behave when applied to different categories of runtime metrics.
- We carry out a sensitivity analysis that explores the impact of varying forecasting horizons on TSF performance.
- We provide a first empirical assessment of two time series foundation models for TSF of software runtime metrics.

4.3.1 Design

In this section, we describe our experiment design, including the dataset, the TSF algorithms we studied, and the specific experimental procedures we used to gather metric predictions.

Dataset

We extend the dataset utilized in our previous study [4] to include a total of 110 distinct software runtime metrics collected from the worldwide IT infrastructure of a large company, which includes tens of thousands of digital devices with heterogeneous hardware and software configurations. For each of the 110 metrics, the dataset provides a time series of daily observations collected over a period of about

one year, thus resulting in a total of 400 measurements per metric. Each metric (*i.e.,* times series) concerns a specific runtime aspect of a particular software application, aggregated across all the devices running that application. For example, one metric might represent the *crash rate* of a specific web browser, while another could indicate the *hang time* of a particular email client. Since the dataset consists exclusively of daily-aggregated values, we could not analyze the variance across all the devices (*e.g.,* the variance may differ between working days and weekends). However, this limitation is discussed in the threats to validity section. The dataset encompasses 29 distinct software applications from diverse typologies, including web browsers, communication platforms, and word processors. The applications have been globally monitored on 7 different runtime aspects, which we refer to as *metric classes*, as illustrated in Table 4.5:

- *Crash rate*: this runtime aspect denotes the average number of observed crashes of an application per hour of use. It is calculated by dividing the total number of observed application crashes within the IT infrastructure by the cumulative hours of application usage across all devices.
- *Hang time percentage*: it represents the percentage of application usage time in which the application remains in a "hang" state, *i.e.,* when the application is unresponsive and not performing any active processing tasks. This runtime aspect can be critical for diagnosing potential issues within software applications.
- *Waiting time percentage*: it is defined as the time users spend in waiting for an application to respond while it is actively running. This metric encompasses periods when the application is unresponsive, known as the "hang" state, as well as application and network are busy loading. It reports the percentage of the total application usage time that is spent in a "waiting" state across all devices within the IT infrastructure.
- *Active time percentage*: it represents the time in which the application is running in the foreground and the user is actively interacting with it, relative to the total application usage time. This metric excludes any application hang time or waiting time from the calculation.
- *Open—Save—Launch time percentage*: all these runtime aspects refer to the time taken for an application to complete the "*open*", "*save*", "*launch*" application activities, divided by the total application usage time. The "*launch*" activity refers to the time required to start the application, while the "*open*" and "*save*" activities refer to the time required for an application to open and save projects/files, respectively.

Table 4.5 shows the number of time series considered in our study for each metric class. It is worth noting that not all metric classes have been considered for the entire set of 29 applications. This is primarily because not every application is monitored across all 7 runtime aspects. For instance, several applications do not produce save time or open time metrics, as their functionalities do not encompass file opening or saving activities. Moreover, we found out that 11 time series contained missing values for some daily observations. Even though having missing values in the collected metrics can be quite common in practice [245], [246], the strategy used for addressing them–such as the imputation with average values or forward-filling by repeating the last observations–would further influence the results. For this reason,

we decided to discard series with missing values. Additionally, we filtered out 10 time series that were constant over time, as this would lead to the trivial forecasting of a flat series. Thus, 21 time series were excluded from the analysis due to missing of constant values.

The time series were provided by the company in a pre-standardized format, using the classic formula outlined in the following equation:

$$z_i = \frac{x_i - \mu_{ts}}{\sigma_{ts}} \tag{4.5}$$

where z_i is the standardized value of the i^{th} data point of the time series, x_i is the original data point value, μ_{ts} is the mean value of the time series measurements, and σ_{ts} is their standard deviation. As shown in Table 4.5, the dataset ultimately consists of 110 time series of 400 values each, comprising a total of 44,000 datapoints[4]

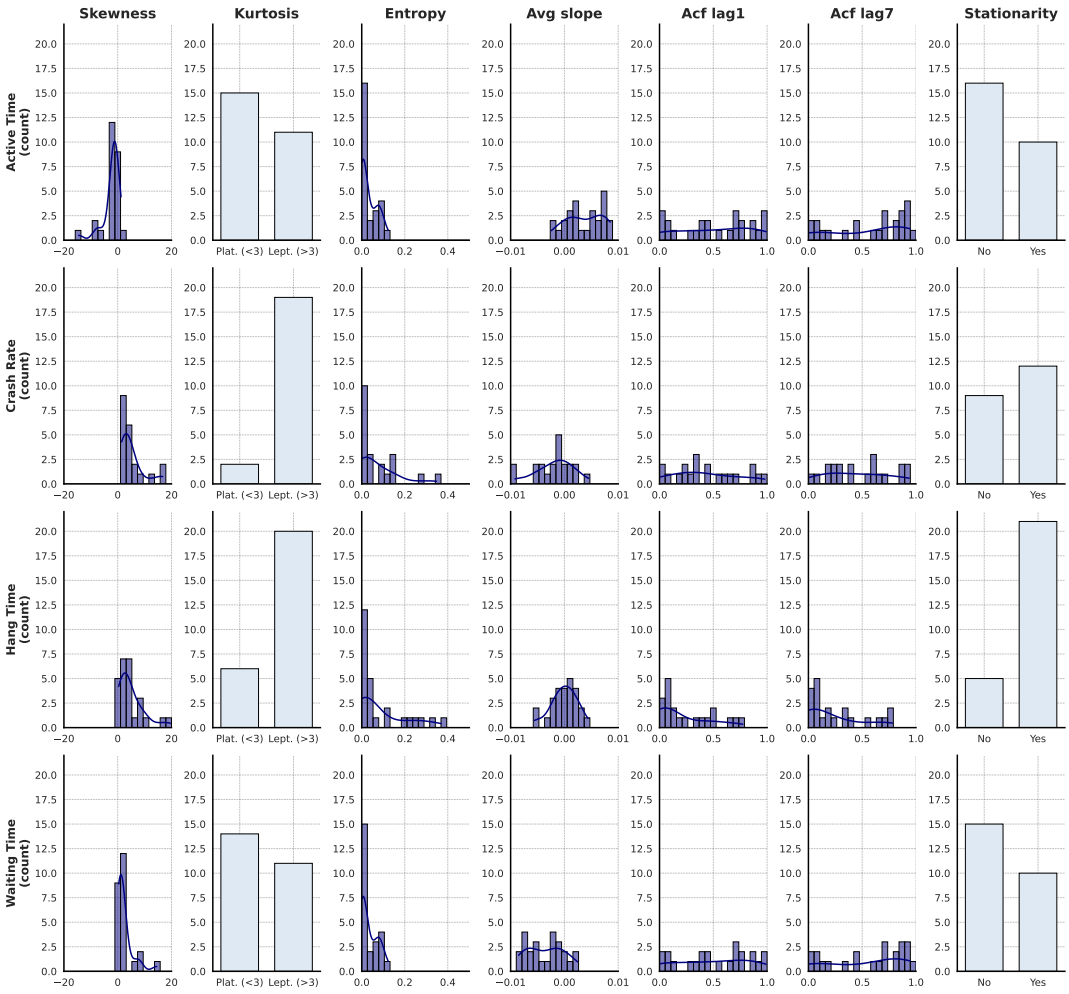


FIGURE 4.8: Time series characterization through several statistical descriptors for each class.

To emphasize the distinct characteristics of each metric class, we provide a qualitative analysis of the time series pertaining to the four major metric classes: Active Time Perc., Crash Rate, Hang Time Perc., and Waiting Time Perc. We excluded the

[4]Due to privacy concerns, we cannot make publicly available the dataset used in our study.

Open-Launch-Save Time Perc. metrics as they contain a limited number of associated series (see Table 4.5). For each time series, we compute the following statistical descriptors: *skewness*, indicating the distribution asymmetry around the mean value; *kurtosis*, characterizing the extreme values of the distribution; *approximated entropy (ApEn)*, which quantifies the regularity of fluctuations over time series data [247]; *autocorrelation (ACF)* at lag 1 and lag 7 (seasonal), indicating the linear dependency between observations with distance 1 and 7 within the series; the absence of linear trends, namely *stationarity*, computed through the AD-Fuller test [235]. We outline the characterization results in Figure 4.8, reporting the distribution of descriptor values grouped by metric class. Kurtosis is reported following the common *platykurtic* (> 3) and *leptokurtic* (> 3) classification, indicating whether the distributions exhibit lighter or heavier tails than a normal distribution. Stationary label is also directly reported based on the AD-Fuller test results ($p < 0.05$). For all the other descriptors, we report the distribution of values using a 20-bin discretization. We observe that there are some visible differences between metric classes. Firstly, the majority of Active Time Perc. time series report a negative skewness, indicating that most of the measurements fall below the mean value, whereas for the other classes we note the opposite tendency. Kurtosis results are more interesting for Crash Rate and Hang Time Perc. metrics, where the leptokurtic behavior is observed for most of the series, indicating heavy tails and extreme values. All the metric classes show an approximated entropy distribution concentrated on low values, indicating limited regularity on the series, despite some minor variations are evidenced between classes. Concerning ACF, we do not observe a strong characterization for specific classes. Stationarity results report that Hang Time Perc. metrics mostly have stationary series (*i.e.*, without trend), while the other classes show a more balanced distribution. Overall, this characterization testifies that time series pertaining to different software metrics classes are likely to have distinct shapes and fluctuation patterns.

TABLE 4.5: Summary of the metric classes, including the total number of time series utilized in the experiment and those discarded during preprocessing.

Metric Class	*Total Time Series*	*Discarded Time Series*
Crash rate	21	8
Hang time %	26	3
Wait time %	25	4
Active time %	26	3
Launch time %	7	2
Open time %	2	1
Save time %	3	0
Overall	**110**	**21**

Forecasting Models

To enhance the representativeness of our study, we selected models from the three principal categories of TSF models, namely classical statistical models, neural networks models, and time series foundation models. For our empirical study, we selected 8 TSF algorithms involving three classical statistical methods (*SARIMA*, *ETS*

and *Prophet*), three neural networks models (*FC-RNN*, *LSTM* and *GRU*) and two time series foundation models (*TimesFM* and *Chronos*).

In the following, we present a concise overview of each model:

Classical Statistical Models

- *Seasonal Autoregressive Integrated Moving Average* (*SARIMA*) [231]: The SARIMA model is an extension of the ARIMA model designed for time series data with seasonal patterns. It incorporates seasonal differencing, seasonal autoregressive terms, and seasonal moving average components to capture repeating patterns and trends.

- *Exponential Smoothing* (*ETS*): The ETS model is a statistical time series forecasting method that decomposes data into error, trend, and seasonality components. It adapts to variations by assigning exponentially decreasing weights to past observations, enabling flexible modeling of various time-dependent patterns.

- *Prophet* [213]: The Prophet model is a forecasting tool that uses an additive approach to model time series data. It accommodates seasonality, holidays, and trend changes, making it robust to missing data and outliers. Its intuitive parameterization enables users to generate accurate predictions with minimal domain expertise.

Neural Networks Models

- *Fully Connected Recurrent Neural Network* (*FC-RNN*) [217], [218]: FC-RNN is a neural architecture where each node in a layer is connected to every node in the subsequent layer, including recurrent connections to manage temporal dependencies. It processes sequential data by maintaining a hidden state that captures information from previous inputs. This model is well-suited for tasks like time series analysis, natural language processing, and sequence prediction.

- *Long Short-Term Memory* (LSTM) [221]: The LSTM model is a type of recurrent neural network (RNN) specifically designed to capture long-term dependencies in sequential data. LSTMs are widely used in tasks like time-series forecasting, natural language processing, and sequence prediction.

- *Gated Recurrent Unit* (GRU) [219]: GRU is a type of recurrent neural network (RNN) architecture designed to efficiently capture long-term dependencies in sequential data. It consists of a reset gate and an update gate, which regulate the flow of information and mitigate the vanishing gradient problem.

Time Series Foundation Models

- *TimesFM* [133]: TimesFM is a pre-trained time series foundation model that demonstrated its effectiveness on a variety of publicly available time series datasets. Specifically, its out-of-the-box zero-shot performance was comparable to supervised models specialized for each individual aspect. In this study, we leverage the 200M parameters release of TimesFM.

- *Chronos* [132]: Chronos is a probabilistic time series model pre-trained on a large collection of publicly available datasets as well as additional synthetic datasets generated via Gaussian process. Chronos demonstrated its effectiveness in different domains to improve zero shot accuracy in forecasting task. Specifically, we adopt the *Chronos-Bolt (Base)* release involving 205M parameters.

Experimental Setup

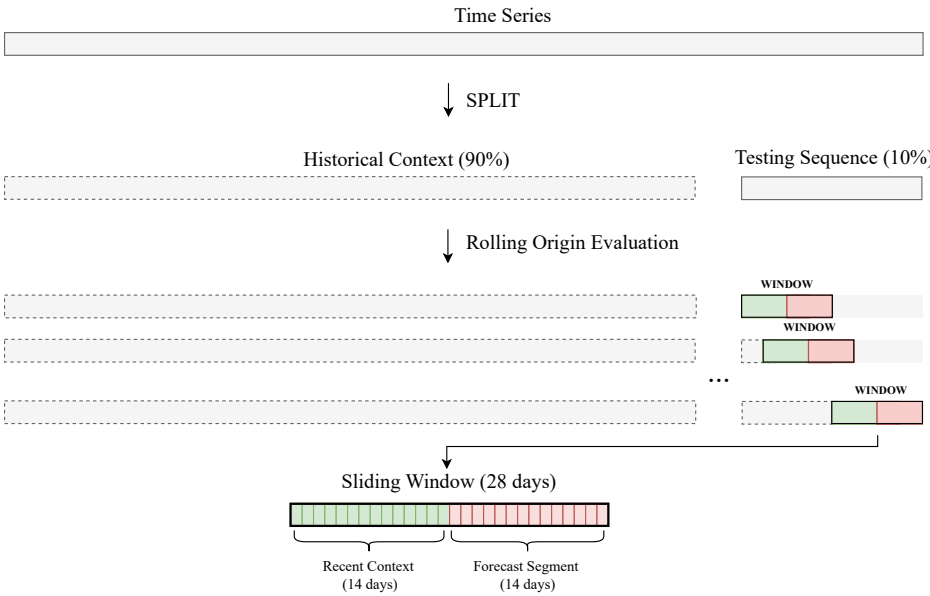


FIGURE 4.9: High-level representation of the evaluation strategy for the forecasting methods. Each time series is initially split into historical context (90%) and testing sequence (10%). Afterwards, the rolling origin cross-validation [238] is applied to the testing sequence, generating testing windows for evaluating each model. Each window includes a recent context part, reported in green, and a forecast segment to predict, reported in red.

We assess the forecasting accuracy of the 8 selected TSF methods across the 110 time series from our dataset. The experiments are conducted independently on each time series. All the forecasting models leverage the entire historical context for generating the predictions. Figure 4.9 provides a visual high-level representation of our experimental setup, which is common to all the forecasting methods. Each method is evaluated in performing *multi-step-ahead* forecasting with a two-weeks forecasting horizon (14 days). All the models use the initial 90% of the sequence (360 measurements) as historical context [232], [248] to capture the time-related patterns associated with each metric, while the remaining 10% (40 measurements) is reserved as testing sequence for evaluating forecasting accuracy. The evaluation is performed leveraging the *rolling origin* strategy [232], [238] for cross-validation. The rolling origin strategy incrementally generates forecasts over the testing sequence by increasing the historical context by 1 time step. The process repeats iteratively until forecasts for the entire testing sequence are generated. To implement this strategy, we leverage sliding-window segmentation [238] across the testing sequence (40 evaluation measurements). As shown in Figure 4.9, each window contains both a

recent context portion, reported in green, and the forecasting segment reported in red, consisting of the ground-truth of the future time steps to generate. Notably, our setup explicitly distinguishes between the historical context and the recent context regions because some methods, *i.e.,* RNN-based ones, need both a training sequence for optimizing model parameters, and a separate recent input context (also known as *look-back* window) to condition the generation of forecasts. Due to this operational constraint, we aligned the evaluation protocol across all models to ensure fairness. We slide a window of 28 consecutive measurements step-by-step, generating 13 overlapping windows for each testing sequence. Within each 28-measurement window, the first half (14 measurements) serves as recent context, and the second half (the subsequent 14 measurements) represents the forecast segment, *i.e.,* the prediction target. Thus, for each window, the model is tasked with predicting a forecast segment F, consisting of the next 14 data points (*i.e.,* the next two weeks of daily measurements) based on the past observations. Consequently, the complete evaluation produces 1,430 forecast segments for each TSF method (13 windows $\times$ 110 time series), resulting to a total of 20,020 individual point forecasts (1,430 segments $\times$ 14 data points). We highlight that our evaluation protocol ensures that all the models are evaluated over the same data, despite the operational constraints posed by each category of models.

In the following subsections, we first introduce the formal notation used throughout the paper. Subsequently, we provide detailed descriptions of the experimental procedures specific to each model category.

Notation

For notational convenience, we use F^i to denote the forecast segment that aims to predict the time series data points beginning from the i^{th} position. For instance, F^{500} represents the forecast segment predicting data from the 500^{th} to the 513^{th} position of the time series. Additionally, we use F_j to denote the j^{th} element of the forecast segment, with $1 \leq j \leq 14$. Figure 4.10 graphically illustrates the example presented above (F^{500}). The black circles represent the data used as input by the models and the red ones are the predicted measurements.

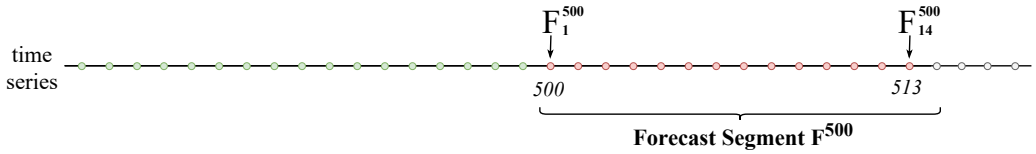


FIGURE 4.10: Forecast segment within the time series.

Note that our evaluation process involves the last 13 forecast segments of each time series. Accordingly, the indices i of the forecast segments F^i used in the evaluation range from 374 to 387 (*i.e.,* $374 \leq i \leq 387$).

Classical Statistical Models

Figure 4.11 reports the detailed experimental process for each category of forecasters. For classical statistical models, we independently create one model instance for each of the 110 time series. In this case, the history used for performing model optimization also constitutes the input data used for generating the predictions. Indeed, statistical models need a historical sequence to perform model fitting (such as coefficients estimation), after which a `forecast()`-like function is invoked to generate

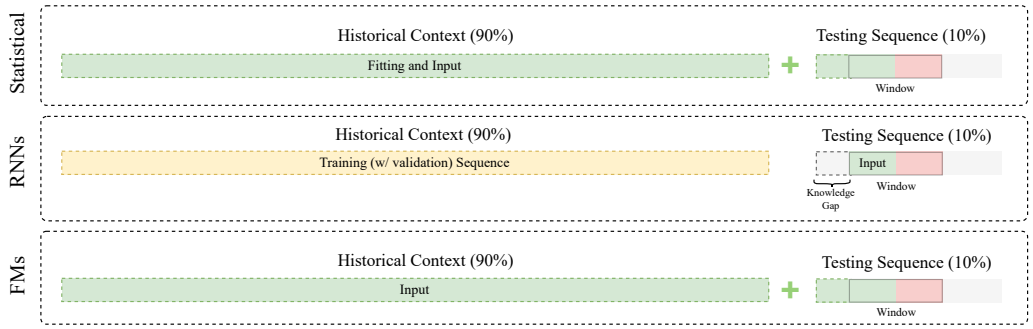


FIGURE 4.11: Evaluation process for each forecasting approach. We report in green the sequence portion used as input by each model. The forecast segments are reported in red. We also highlight in yellow the portion of the series that is used during the RNNs training process for optimizing the weights.

future steps, without explicitly specifying recent context as input; it is supposed to be included as last part into the fitting sequence. For this reason, the fitting data is composed of the historical context concatenated with the recent context, reported in green in Figure 4.11. Concerning the SARIMA model, $(p, d, q)(P, D, Q)_s$ parameters must be defined before performing the model fitting, where: p and P represent the order of the autoregressive (AR) terms for the non-seasonal and seasonal parts respectively; d and D denote the degree of differencing needed to account for trends and render the series stationary on both non-seasonal and seasonal levels; q and Q indicate the order of the moving average (MA) terms for the non-seasonal and seasonal components; and s denotes the seasonality period of the time series data. To estimate the d and D parameters, we considered the number of times we applied differencing to obtain a stationary time series according to the ADFuller [235] test. Although the stationarity assumption of SARIMA model, it supports deterministic and stochastic trends by automatically applying differencing operations when the orders are greater than zero. The best values of (p, q, P, Q) have been searched by fitting the model with all the values ranging from 0 to 3 for each parameter, and by selecting the parameter configuration that gave the lowest *Akaike Information Criterion* (AIC) [236], [237] value after the model fitting. Specifically, our parameter search exhaustively evaluates all the possible configurations in that range, using the historical context as fitting sequence. The range for the optimized parameters search was defined to be relatively narrow (*i.e.*, 0–3), according to the standard practice in the field [238], [249], [250]. Indeed, as the search range widens, the number of possible parameter configurations grows very quickly, leading to a significant increase of the search time. To fit the SARIMA model, we employ the *L-BFGS* [251] solver as the default. In cases where numerical instability led to calculation exceptions, the *Powell* method [252] was employed as an alternative. We set the s parameter to 7, which corresponds to a one-week seasonality. The choice of this value has been supported by industrial stakeholders input and by manual inspection of a subset of time series exhibiting pronounced weekly seasonal patterns. Concerning the ETS model, we employ Seasonal-Trend decomposition using LOESS [253] to estimate the trend type (*i.e.*, additive or multiplicative). Prophet model does not require parameters specification.

Finally, during the evaluation phase, we use the fitted and parameterized models to generate the 13 forecast segments F for each time series. Specifically, when predicting a particular forecast segment F^i, the model is fitted using the entire historical

sequence of measurements available up to the start of that forecast segment (*i.e.*, all measurements preceding the segment F^i). In Figure, 4.11 we report in green the historical input sequence and in red the forecast segments. This methodology enables us to simulate a realistic scenario of progressive forecasting, in which the model is continuously tested as new data becomes available in the time series.

As a result, we obtain 1,430 forecast segments per statistical model (13 forecast segments $\times$ 110 time series).

Neural Network Models

The neural network models selected for this study are based on the recurrent neural network (RNN) architecture, specifically designed for sequence-related tasks. RNNs, like many other ML time-series models, learn a mapping from the historical recent context, used as input window, to the subsequent future segment for generating forecasts, leveraging a large set of historical examples. To evaluate a RNN model, we independently train one model instance for each of the 110 time series, by still using the initial 90% of time series measurements for training, and the remaining 10% for evaluation, as reported in Figure 4.9. To guide training, we set aside the last 10% of the training sequence as validation data. Following the established practice in the literature, *sliding window segmentation* is applied to the training sequence as well (including validation set). We use a sliding window of 28 consecutive measurements to generate multiple overlapping segments. The initial 14 measurements serve as the model input (*look-back* period), whereas the remaining 14 measurements form the prediction target used for training. Worth noting that choosing different window sizes may yield different results. We set the training window length taking into account the number of windows available for the training process of RNNs. Indeed, increasing the window length reduces the number of training instances. For instance, extending the window length to 360 datapoints would result in a total of only 40 training windows for each time series. Our assumption is that using a recent context of two seasonal cycles (*i.e.,* 14 days) provides a balanced number of training samples and input data. In Figure 4.11, we summarize this process on the left panel of RNNs evaluation, showing inputs in green and the training prediction outputs in red. Notably, even though RNN models are trained using a sliding window setting, they are still optimized over the entire training and validation sequences (consisting of 360 values). Indeed, the forecast segments are generated after being trained over the entire historical context. The yellow portion in Figure 4.11 represents the historical context data that RNNs benefit from at the end of the training stage. It can be observed that all the historical context is used at the end of the process. In green, is reported the data input used to build training samples, composed of 14 observations. The architecture of RNN-based models under consideration (*i.e., FC-RNN*, *LSTM*, *GRU*) is designed as follows. The first layer of the architecture consists of 14 units of the specific RNN type (either *FC-RNN*, *LSTM*, or *GRU*). The architecture includes a second layer composed by other 14 units of the same type, which is encapsulated within a bidirectional layer [254]. Finally, the processed information is channeled through a fully connected layer with 14 units, by employing the hyperbolic tangent (*tanh*) activation function to produce the final output. We use the Adam optimizer [192] 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. Additionally, we implement a reduce-on-plateau strategy, which decreases the learning rate when no improvement is observed in the validation loss for 20 epochs. The training is conducted for a maximum of 500 epochs, with an early stopping mechanism that

terminates the process if no improvement in the validation loss is observed for 50 consecutive epochs. The best model is selected basing on the lowest validation loss observed throughout the training epochs. An alternative evaluation strategy would be to employ *online training* [255], [256] to adapt the RNN models in real-time during the evaluation stage. In this way, they also use the time series values located between the training series (*i.e.,* the initial 90%) and the current forecasting segment to predict. When using online training, we observed only marginal performance changes among the two methods, without affecting our findings. This behavior can be attributed to our experimental setting, where each time series results in 40 testing data points employing a sliding window of 28 measurements. Consequently, the widest effective adaptation range with the online training spans 12 measurements, which constitutes a relatively limited amount of new information. However, we decided to report the results for the offline training setting since online training is usually expensive in software monitoring domain. We report the online training results in our replication package[5].

As a result of the evaluation process, similarly to statistical models, we obtain for each RNN model a set of 1,430 forecast segments (13 $\times$ 110 time series).

Time Series Foundation Models

We leverage the zero-shot forecasting capability of foundation models by providing the historical measurements series as a prompt and requesting to generate the subsequent 14 values (*i.e.,* the forecast segment F), without performing any fine-tuning.

Figure 4.11 reports in the lower panel the evaluation methodology used for foundation models. These models just need a single historical sequence to generate forecasts, that we build concatenating the historical context with the recent context. Specifically, when predicting the forecast segment F^i, we provide as input the first $i-1$ measurements of the time series (*i.e.,* all observations preceding F^i). This evaluation methodology follows the same procedure applied in statistical models. In our setup, foundation models do not work with a textual prompt as typically done with language models; instead, the input consists of a numerical array containing the historical sequence itself.

Aligning the evaluation with the other models, we set the forecasting *horizon length* for both *TimesFM* and *Chronos* to 14, by using their default other parameters. Additionally, the batch size per core for the *TimesFM* model is configured to 32. We report the specific version of the library implementing each model in §**??**. To account for a fair comparison among models, we only considered *Chronos* predicted mean values without considering the probabilistic distribution of predictions, to ensure a single predicted data point per day. Even though these models do not currently support explicit seed configuration, we fixed the random seed of the underlying libraries (*e.g., torch* and *numpy*) to a fixed value (*i.e.,* 42 in our case), and manually verified that the predictions remained constant when repeatedly executing few sample runs.

Similar to the other models, this process results in a total of 1,430 forecast segments per model (13 $\times$ 110 time series).

[5]Our replication package is available at the following link: `https://github.com/SpencerLabAQ/replication-package__tsf-ext.git`

Implementation details

All the experiments have been implemented in *Python*. We utilized the *statsmodels*[6] library (vers. *0.14.4*) for implementing the SARIMA and ETS models. The *Prophet* model has been utilized following the instructions reported in the official release[7] (vers. *1.1.6*). Neural network models have been implemented using `SimpleRNN` (for FC-RNN models), `LSTM` and `GRU` classes of *Tensorflow*[8] (vers. *2.28*). For the foundation models, we employed the `timesfm-1.0-200m-pytorch`[9] (implemented using *timesfm 1.2.3*[10]) and `chronos-bolt-base`[11] (with *chronos-forecasting 1.4.1*[12]) checkpoints using the *PyTorch*[13] library (vers. *2.5*).

Evaluation Metrics

Typical evaluation metrics for TSF approaches consider the residual difference between the predicted values and the ground-truth. Typical scale-dependent measures are Mean Absolute Error (MAE) [75], [135], Mean Squared Error (MSE) [55], and Root Mean Squared Error (RMSE) [5], [56], which report the error values in the original dataset scale emphasizing larger errors in different ways. Percentage-based metrics, such as *Mean Absolute Percentage Error* (*MAPE*) [56], [74] and *Symmetric Mean Absolute Percentage Error* (*SMAPE*) [4], [29], are also widely employed in this context to report the error relatively to the ground-truth values. Another well-established and scale-independent evaluation metric is *Mean Absolute Scaled Error* (MASE) [132], [232], and it consists in normalizing the absolute errors using the predictions made with a naive approach (typically replicating the last observations).

Mean Absolute Error (MAE)

The experiments were conducted using data that had already been standardized. Therefore, we selected *Mean Absolute Error* (*MAE*) as the primary metric for evaluating the forecasting accuracy of TSF models. *MAE* measures the absolute difference between the predicted and standardized actual values, offering clear interpretability. We did not further scale the error (*e.g.,* using MASE) because in our setting, involving standardized series, the MAE is already scale-independent. In addition, the comparison against the naïve approach is explicitly reported in RQ1.

In contrast, percentage-based metrics, such as *Mean Absolute Percentage Error* (*MAPE*) and *Symmetric Mean Absolute Percentage Error* (*SMAPE*), are less suitable because standardization often reduces interpretability and inflates error values, particularly when the standardized values are near zero. For example, a predicted value of 0.01 versus an actual value of 0.005 would yield a 100% error, despite the small absolute difference.

Similarly, *Mean Squared Error* (*MSE*), emphasizes larger errors by squaring deviations. While this characteristic can be advantageous in certain situations, it may exaggerate moderate deviations in standardized data, leading to misleading assessments.

[6] https://www.statsmodels.org/stable/index.html
[7] https://github.com/facebook/prophet.git
[8] https://www.tensorflow.org/
[9] https://huggingface.co/google/timesfm-1.0-200m-pytorch
[10] https://pypi.org/project/timesfm/
[11] https://huggingface.co/autogluon/chronos-bolt-base
[12] https://pypi.org/project/chronos-forecasting/
[13] https://pytorch.org/

MAE, being linear, evaluates all deviations equally, making it a balanced and robust metric for standardized data scenarios. Notably, MAE in standardized contexts directly relates to the standard deviation (σ) of the original data. For example, a *MAE* of 0.5 indicates that predictions deviate, on average, by half a standard deviation from the actual values. The MAE has been extensively used for evaluating time series forecasting approaches [56], [133], [135], [257].

Average Rank

Instead of focusing solely on forecasting accuracy, we also report the *average rank* of each TSF model. Specifically, for each time series, we rank the models based on their MAE, assigning a rank of 1 to the most accurate model and 8 to the least accurate. We then compute the *average rank* of each model across the 110 considered time series. This metric provides a measure of the model forecasting accuracy relative to other models. In Section 4.3.2, we provide a detailed explanation of the methodology employed to calculate and evaluate these values for each research question.

4.3.2 Results

$RQ_{3.1}$. Are the selected TSF methods suitable for predicting software runtime metrics?

Motivation. This RQ serves as a sanity check to evaluate whether the extended set of TSF models selected for the advanced study are suitable for predicting software runtime metrics. The preliminary study already provided empirical evidence of the superiority four TSF methods with respect to two straightforward baselines. Here, we extend this investigation to a wider set of forecasting methods, used in subsequent analyses, to verify whether this result generalizes across a larger dataset.

Methodology. To address this RQ, we compare the forecasting accuracy of TSF models with that of two naive baseline methods, which predict the future values by employing simple statistical heuristics, based on past historical values, namely seasonal naive (*sNaïve*) and seasonal monthly mean (*sMM*) [4]. *sNaïve* operates under the assumption of temporal recurrence, by repeating the last observed values for future forecasts. In other words, this method projects the most recent seasonal data forward to the next equivalent season, thus assuming cyclical repetition. For instance, when predicting values for the upcoming week, the method simply replicates the observations from the previous week. This method is commonly used as baseline for comparative analysis in TSF studies [54], [205], [214], [232], [233], [234]. *sMM* leverages recent seasonal weekly trends to forecast future values. Specifically, it computes predictions by calculating the mean of data points that correspond to the same weekday in the preceding month. For example, to forecast the value for an upcoming Monday, sMM would average the values from all Mondays in the last month. This baseline reflects analytical processes employed by software monitoring teams in practice [4].

First, we generate 1,430 forecast segments for each model and baseline. We then compute their *MAE* across all time series. In this RQ, we assess the forecasting accuracy of each model in a "*one-step-ahead*" setting—namely, evaluating each model ability to predict the next data point in the time series (*i.e.*, F_1). We adopt this setting because it is commonly used in TSF studies [54], [232], [240], [241]. However, we will also evaluate the "*multi-step-ahead*" capabilities of the models in a subsequent research question ($RQ_{3.4}$).

Formally, given a time series Y and a particular TSF method, we calculate the corresponding *MAE* as follows:

$$\text{MAE} = \frac{1}{n-k+1}\sum_{t=k}^{n}\left|F_1^t - Y_t\right| \tag{4.6}$$

where k and n denote the positions of the first and last elements of the time series used for evaluation. F_1^t denotes the first element of the forecast segment F^t (*i.e.*, the next-day prediction of the TSF method for the t^{th} element of the time series), and Y_t denotes the actual time series measurement at position t. As a result of this process, for each TSF model and baseline, we obtain 110 MAE results, *i.e.*, one per time series.

We assess the forecasting accuracy of TSF methods against naive baselines by comparing their mean and median *MAE* values. Additionally, to evaluate whether the observed differences are statistically significant, we conduct a Friedman ANOVA [258] followed by post-hoc Wilcoxon signed-rank tests [157] with Holm-Bonferroni correction [259]. The Friedman ANOVA is used to identify whether statistically significant differences exist among the methods. The post-hoc Wilcoxon signed-rank test is then used to pinpoint where these differences lie between specific *model-baseline* pairs, while the Holm-Bonferroni correction is applied to adjust for multiple comparisons and control the Type I error rate.

TABLE 4.6: $RQ_{3.1}$. Comparison of TSF models and both sNaïve-sMM baselines. The table reports the percentage differences in mean and median MAE values, along with the results of the post-hoc Wilcoxon signed-rank tests (p-value).

	sNaïve			*sMM*		
	p-value	mean	median	p-value	mean	median
SARIMA	< 0.001	-30%	-26%	< 0.001	-11%	-10%
ETS	< 0.001	-29%	-23%	< 0.01	-10%	-6%
Prophet	-	-16%	-9%	-	+6%	+11%
FC-RNN	< 0.001	-27%	-24%	-	-7%	-6%
LSTM	< 0.001	-24%	-19%	-	-3%	-1%
GRU	< 0.001	-27%	-23%	-	-7%	-6%
TimesFM	< 0.001	-33%	-29%	< 0.001	-14%	-13%
Chronos	< 0.001	**-34%**	**-30%**	< 0.01	**-16%**	**-15%**

Results. Table 4.6 presents the results of the comparison between TSF models and baseline methods[14]. The results reveal that TSF models consistently outperform the sNaïve baseline, achieving mean *MAE* reductions ranging from 16% (Prophet) to 34% (Chronos). Median *MAE* improvements similarly span from 9% to 30%. When compared to the sMM baseline, TSF models also demonstrate improved mean and median *MAE* values, though the margins of improvement are more modest. Specifically, only SARIMA, TimesFM and Chronos surpass a 10% reduction in both mean and median *MAE*, whereas the other TSF models achieve mean reductions between 3% and 10%. Notably, Prophet is the sole TSF model displaying higher *MAE* than the sMM baseline.

[14]Notably, although the MAE differences are expressed in percentage terms, this approach differs from reporting the MAPE values, which consider the point-wise percentage differences within the predicted windows instead.

To evaluate the statistical significance of these differences, we employed Friedman ANOVA followed by the Wilcoxon signed-rank test as a post-hoc analysis. The Friedman ANOVA yields a p-value < 0.001, confirming statistically significant differences across the MAE values. The subsequent post-hoc analysis indicates significant differences between most of the TSF models and the sNaïve baseline, with p-values consistently < 0.001, highlighting a clear and statistically significant advantage in forecasting accuracy over sNaïve. The only exception is Prophet, that does not report statistical significance. Improvements from TSF methods relative to the sMM baseline are less pronounced, underscoring the strength of sMM as a baseline. Among TSF models, only Chronos, TimesFM, SARIMA, and ETS achieve statistically significant superior performance compared to sMM, with respective p-values of < 0.001, < 0.001, < 0.001, and < 0.01.

Answer to RQ$_{3.1}$: All TSF models except Prophet outperform the sNaïve baseline with very high statistical significance ($p < 0.001$). SARIMA, ETS and foundation models outperform the sMM baseline with high significance ($p < 0.01$). These findings support the suitability of the selected TSF models for predicting software runtime metrics.

RQ$_{3.2}$. Which TSF models provide the most accurate predictions of software runtime metrics?

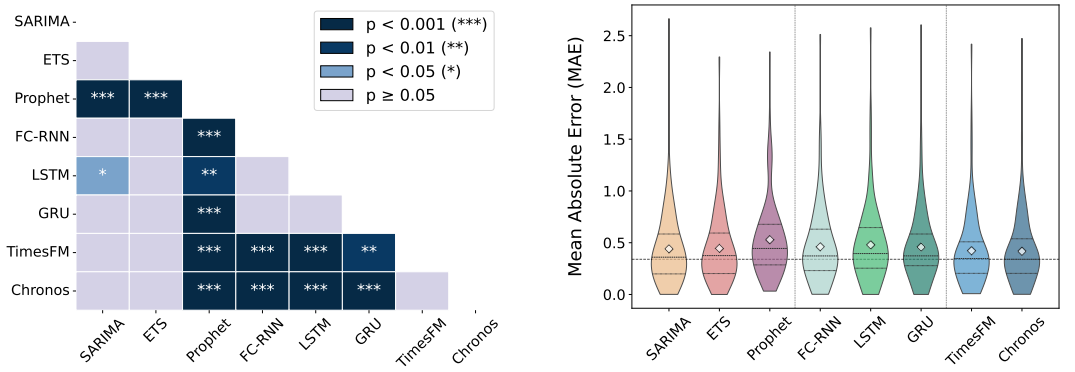
Motivation. After verifying the general suitability of TSF models for predicting software runtime metrics, we aim to identify the best performing models. Determining the most effective TSF models in this context will help practitioners and researchers make more informed choices when integrating TSF into their software engineering processes.

Methodology. To address this RQ, we first compute MAE for each model in a "*one-step-ahead*" setting, using the same approach employed in the previous research question. Thereafter, we compute the average rank following the procedure presented in Section 5.4.2. Specifically, given R_{mi} as the rank of the model m in forecasting the i-th time series (where lower rank indicates better accuracy) and n as the number of time series considered (*i.e.,* 110), we compute the average rank of each model as:

$$\text{Avg. Rank}_m = \frac{1}{n} \sum_{i=1}^{n} R_{mi} \tag{4.7}$$

To compare the forecasting accuracy of different TSF models, we analyze both their average ranks and MAE distributions. In addition, we perform a rigorous statistical assessment to determine the significance of observed differences. Specifically, we apply Friedman ANOVA followed by post-hoc Wilcoxon signed-rank tests with Holm-Bonferroni correction.

Results. We report our results in Figure 4.12. The heatmap (Figure 4.12a) presents the p-values obtained by comparing each pair of forecasting approaches using the Wilcoxon signed-rank test, while the violin plots of Figure 4.12b report the MAE distributions of each forecasting model. These two figures provides a comprehensive visualization of which TSF model performed better or worse than the others. For



(A) Heatmap of p-values obtained from the post-hoc Wilcoxon signed-rank test, illustrating the statistical significance of differences in MAE between model pairs.

(B) Distribution of MAE values for the various TSF models across the time series.

FIGURE 4.12: $RQ_{3.2}$. Comparison of TSF models in terms of MAE distribution and statistical significance of differences.

instance, if we want to compare FC-RNN against Chronos, Figure 4.12a shows that the two approaches exhibit statistically significant differences in their performance (three asterisks indicate that the corresponding p-value is lower than 0.001, as reported in the figure legend). To understand which model performs better, we wefer to Figure 4.12b, where we see that the error distribution associated with FC-RNN is shifted toward higher values relative to that of Chronos. Thus, we conclude that Chronos performed significantly better than FC-RNN, supported both by statistical test results and the distribution shift. The violin plots illustrate that foundation models (*i.e.*, TimesFM and Chronos) exhibit lower MAE values compared to other models, indicating higher forecasting accuracy. In particular, TimesFM and Chronos achieve the lowest median MAE values, 0.35 and 0.34, respectively. We observe that both statistical and neural network models exhibit similar MAE distributions, except for Prophet, which visibly shows higher mean and median error values.

To further assess differences in forecasting accuracy among various TSF models, we performed a Friedman ANOVA followed by a post-hoc Wilcoxon signed-rank test on the MAE distributions. The Friedman ANOVA yielded a highly significant result ($p < 0.001$, *5.03e-16*), indicating statistically significant differences among the MAE distributions. Given this outcome, we applied the Wilcoxon signed-rank test pairwise to identify specific differences between models. The results of the Wilcoxon test are visualized in the heatmap presented in Figure 4.12a, where darker colors represent higher statistical significance. We notice that foundation models (*i.e.*,, TimesFM and Chronos) significantly outperform FC-RNN, GRU, and Prophet, with very high statistical significance ($p < 0.001$). LSTM is still outperformed by Chronos (with $p < 0.001$), although the comparison with TimesFM shows lower statistical significance ($p < 0.01$). The only cases where the differences between foundation models and another model are not statistically significant are with ETS and SARIMA. Additionally, SARIMA outperforms LSTM with statistical significance ($p < 0.05$), and Prophet is outperformed by all the other models with high significance ($p < 0.001$ in most of the cases). Nevertheless, as seen in Figure 4.12b, Chronos and TimesFM both achieve lower median MAE values than ETS and SARIMA.

The results presented in Table 4.7, which summarizes the average ranks of the

TABLE 4.7: RQ_2. Average rank of each TSF model across the time series. The best-performing model (*i.e.*, lowest rank) is **bolded**, and the second-best is underscored.

	Statistical			Neural Networks			Pretrained	
	SARIMA	*ETS*	*Prophet*	*FC-RNN*	*LSTM*	*GRU*	*TimesFM*	*Chronos*
Avg. Rank	3.93	4.22	6.00	4.93	5.09	4.58	3.75	**3.51**

TSF models, further support the superior forecasting accuracy of foundation models over other approaches. Specifically, Chronos and TimesFM attain the lowest average ranks, with values of 3.51 and 3.75, respectively.

Although the average ranks for Chronos and TimesFM are lower than those of the other models, they remain considerably higher than the optimal average rank of 1. This indicates that foundation models do not systematically outperform other models across all time series. To further investigate these observations, we analyzed how frequently each model ranked among the top three positions, as shown in Figure 4.13. Specifically, the figure illustrates the frequency with which each model attained one of the top-three rankings across the time series. Firstly, we observe that

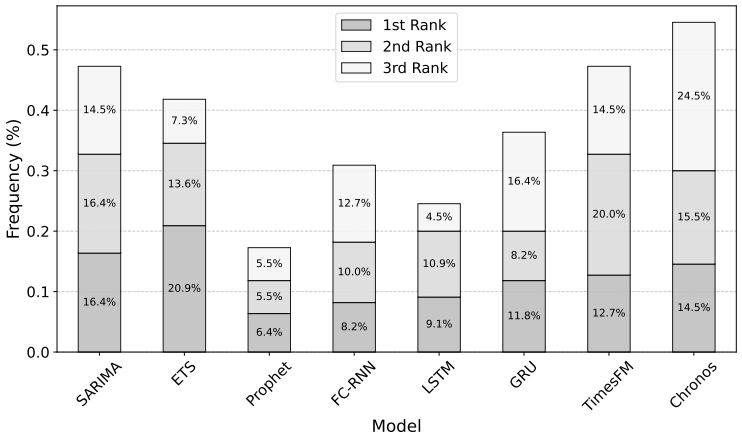


FIGURE 4.13: $RQ_{3.2}$. Distribution of top-three rankings. For each TSF model, the plot shows the percentage of time it placed 1st, 2nd, or 3rd across the time series.

all TSF models achieved the best ranking on a considerable proportion of time series (ranging from 6.4% to 20.9%), and frequently secured top-three rankings (ranging from 17.4% to 54.5%). This suggests that the choice of the optimal model heavily depends on the specific characteristics of each time series. Interestingly, ETS obtained the highest frequency of best rankings (20.9%), despite its relatively lower average rank (4.22). In other words, although ETS performs poorly compared to other models on average, it can still deliver the best predictions for a substantial subset of the time series. These findings highlight the importance of carefully selecting TSF models based on specific time series. While foundation models generally offer consistent forecasting accuracy across diverse time series, other models can still yield superior predictions for particular series. Therefore, practitioners aiming for optimal accuracy should consider evaluating multiple TSF models rather than relying exclusively on a single approach.

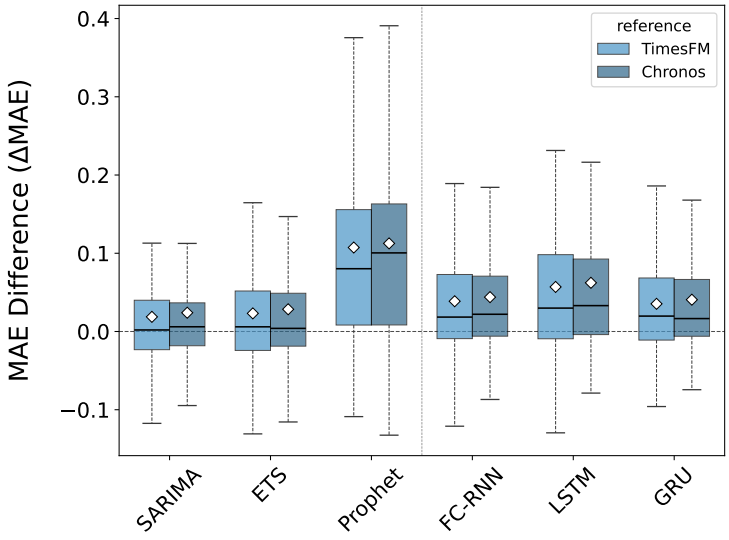


FIGURE 4.14: $RQ_{3.2}$. Box plots of paired ΔMAE relative to the foundation models (TimesFM and Chronos); negative values indicate lower error (better performance) than the reference model.

Based on the previous results, we see that foundation models provide on average lower errors and higher ranks compared to the other approaches. In addition to average ranks and error distributions for each model, we isolate the foundation models as a reference group and we compute the pairwise difference between their MAE and the MAE of all the other approaches. Results are outlined in Figure 4.14, where positive values indicate that, given a time series, the error of the model under analysis is higher than the error of TimesFM or Chronos. The figure shows that the distribution of error differences is skewed towards positive values indicating, as expected, that for most of the series foundation models outperform the other approaches on average. Prophet reports the widest distribution with very high mean values compared to the others, also reporting the first-quartile values above zero, indicating substantially lower accuracy compared to foundation models. Instead, the error difference distribution of SARIMA and ETS models report a median quite close to zero against both TimesFM and Chronos, with a significant number of negative values. This indicates that, for a significant amount of time series, they are able to deliver superior performance compared to foundation models.

Answer to $RQ_{3.2}$: Foundation models achieve the lowest average ranks (3.51 and 3.75), significantly outperforming most of comparator models ($p < 0.05$). However, they do not consistently deliver the best predictions across all time series. For instance, ETS, despite its relatively low overall forecasting accuracy, still produces the best forecasts for a notable proportion (20.9%) of the analyzed time series.

$RQ_{3.3}$. Do TSF models exhibit diverse forecasting accuracy over different classes of software runtime metrics?

Motivation. The third research question aims to analyze the forecasting accuracy of TSF methods for different classes of software runtime metrics that, as reported

in Figure 4.8, tend to exhibit different fluctuation patterns. Motivated by the outcomes of $RQ_{3.2}$, which indicated that forecasting accuracy significantly varies across individual time series, here we pursue two distinct research objectives: (i) to determine whether certain TSF models are more effective than others for specific metric classes, and (ii) to investigate whether the general performance of TSF models varies significantly across different metric classes.

Methodology. Similar to the previous two research questions, we assess the forecasting accuracy of TSF models using MAE in a *one-step-ahead* setting (see Equation 4.6). However, rather than analyzing MAE values in aggregate, we group them based on the metric class of each time series. The average rank is computed in the same manner as in the previous research question (see Equation 4.7), but limited to time series within the same class. The evaluation considers both the MAE distributions and the average ranks for each forecasting model. Out of the seven metric classes considered in this study, *Save Time Perc*, *Open Time Perc*, and *Launch Time Perc* are excluded from the analysis due to the limited number of associated time series (see Table 4.5).

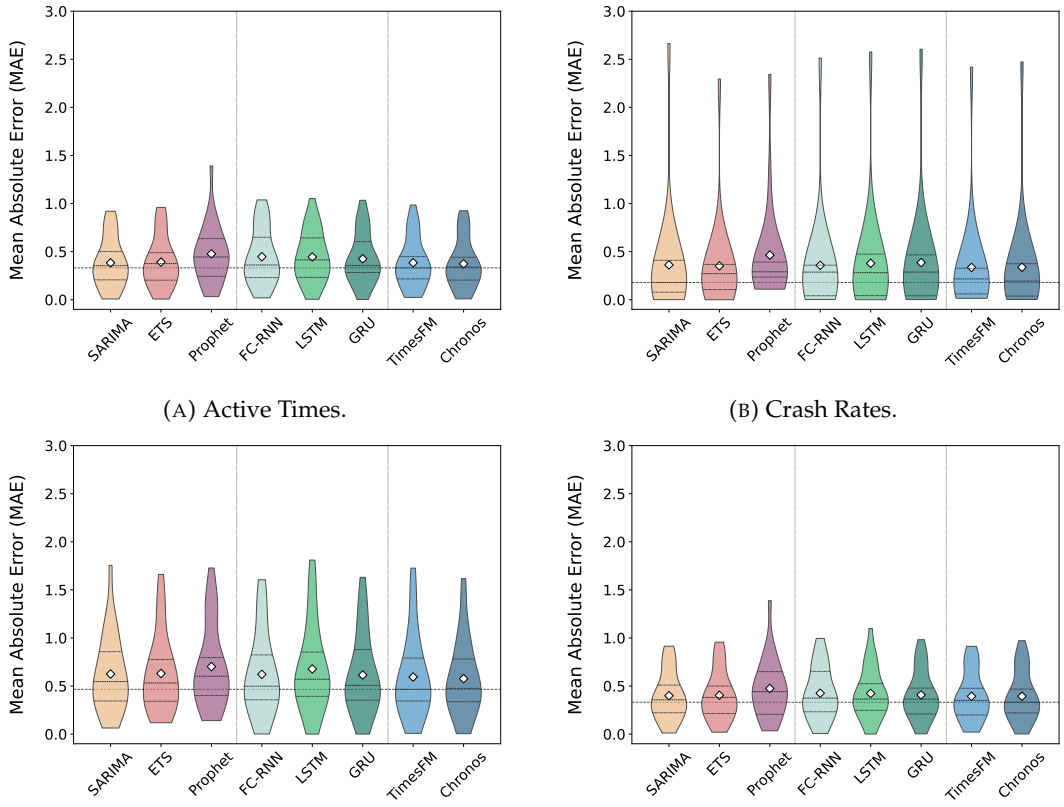
For the first research objective—determining whether specific TSF models are more effective than others for particular metric classes—we assess the significance of differences in MAE values among TSF models within each metric class. Specifically, we employ Friedman ANOVA followed by post-hoc Wilcoxon signed-rank tests with Holm-Bonferroni correction.

For the second research objective—investigating whether the overall performance of TSF models significantly differs among metric classes—we compare the MAE values produced by the TSF models across different metric classes. This analysis aims to reveal whether certain metric classes are inherently more challenging to forecast than others. To accomplish this, we utilize the Kruskal-Wallis test [260] followed by post-hoc Mann-Whitney U-tests [261] with Holm-Bonferroni correction.

Results. The MAE distributions for each metric class are depicted in Figure 4.15. We observe that foundation models generally exhibit lower medians across all metric classes. The only exception is for Crash Rate class, where SARIMA achieved the lowest median value. The average ranks for each metric class are reported in Table 4.8. Chronos and TimesFM consistently achieve the top-two ranks, except for the *Active Time Perc.* metric class, where Chronos achieves the best rank and SARIMA the second-best, and *Waiting Time Perc.* metrics, where SARIMA achieved the best rank followed by Chronos. The superiority of SARIMA in forecasting waiting times corroborates our observations from our previous study [4], which identified SARIMA as the top-performing model over waiting time series.

Table 4.9 reports the results of Friedman ANOVA for each metric class. We find statistically significant differences ($p < 0.05$) for the *Active Time Perc.*, *Hang Time Perc.*, and *Waiting Time Perc.* classes. This indicates that, for these metrics, forecasting accuracies reported by the TSF models significantly differ from each other. To better understand these differences, we apply the post-hoc Wilcoxon signed-rank test. The results[15] are presented in Figure 4.16. For the *Active Time Perc.* class (Figure 4.16a), we observe high statistical significance when comparing Prophet with both foundation models and SARIMA. In addition, statistically significant difference appears between Chronos and FC-RNN. Despite visible differences in mean and median MAE values, this result suggests that no model consistently outperforms others

[15]It's worth noting that we only report results for metric classes where Friedman ANOVA led to statistically significant results.



(A) Active Times.

(B) Crash Rates.

(C) Hang Times.

(D) Waiting Times.

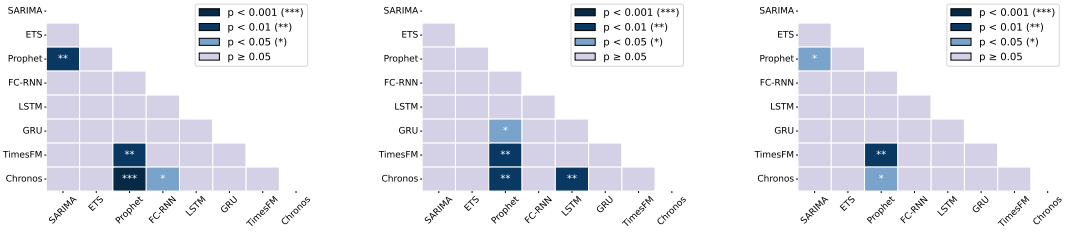
FIGURE 4.15: $RQ_{3.3}$. MAE values of the various TSF models grouped by metric class. Classical statistical models are shown in red, recurrent neural network (RNN) models in green, and foundation models in blue. For each class, a horizontal dashed line indicates the best median MAE value as a reference.

TABLE 4.8: RQ_3. Average rank of TSF models grouped by metric class. The best-performing model in each row is **bolded**, and the second-best is underscored. Rows with statistically significant differences (*i.e.*, $p < 0.05$, according to Friedman ANOVA) are highlighted.

	Statistical			Neural Networks			Pretrained	
	SARIMA	*ETS*	*Prophet*	*FC-RNN*	*LSTM*	*GRU*	*TimesFM*	*Chronos*
Active Time Perc.	3.46	4.12	6.12	5.65	5.19	4.35	3.92	**3.19**
Crash Rate	4.19	4.33	5.67	4.33	4.62	5.14	4.00	**3.71**
Hang Time Perc.	4.77	4.42	5.88	4.50	5.23	4.08	3.65	**3.46**
Waiting Time Perc.	**3.60**	4.16	6.04	5.28	5.00	4.36	3.80	3.76

TABLE 4.9: RQ_3. Friedman ANOVA results for each metric class. Rows with statistically significant results ($p < 0.05$) are highlighted.

Metric Class	Friedman ANOVA Statistic	*p-value*
Active Time Perc.	**<0.01**	**Yes**
Crash Rate	0.199	No
Hang Time Perc.	**<0.05**	**Yes**
Waiting Time Perc.	**<0.05**	**Yes**



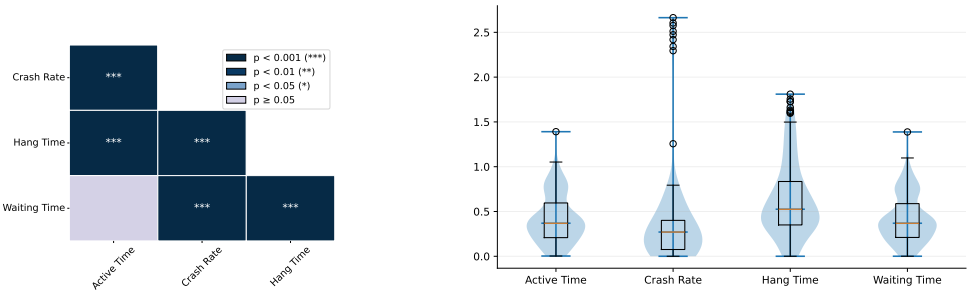
(A) Active Times Perc. (B) Hang Times Perc. (C) Waiting Times Perc.

FIGURE 4.16: $RQ_{3.3}$. Heatmap of p-values obtained from the post-hoc Wilcoxon signed-rank test, illustrating the statistical significance of differences in MAE between model pairs within three specific metric classes: Active Times Perc., *Hang Times Perc.* and *Waiting Time Perc.*

within this metric class, aside from Prophet. For the *Hang Time Perc.* class, we observe more statistically significant differences, particularly in comparisons involving foundation models against SARIMA, Prophet, and LSTM. Among traditional statistical models, both SARIMA and ETS demonstrate significantly better accuracy compared to Prophet. Regarding Waiting Times Perc. class (Figure 4.16c), only Prophet reported statistically significant difference compared with SARIMA, TimesFM and Chronos.

Based on these findings, we conclude that foundation models generally remain the most effective choices irrespective of the specific metric class, although they are outperformed by SARIMA in terms of average rank on Crash Rate series. Nevertheless, the relative forecasting accuracy of individual TSF models can vary notably depending on the particular metric under consideration. For example, by examining Figure 4.15d, we find that all TSF models exhibit similar MAE distributions for *Waiting Time Perc.* metrics, whereas for other metrics, such as *Hang Time Perc.*, model performance diverges substantially (see Figure 4.15c). Notably, SARIMA achieves the best average rank for *Waiting Time Perc.*, yet exhibits the third-worst rank for *Hang Time Perc*. We hypothesize that this behavior could stem from the SARIMA capabilities to capture regular seasonal patterns. Indeed, such patterns might be more prevalent in metrics closely related to regular software usage, such as *Waiting Time Perc.*, compared to those metrics that more directly reflect unexpected software issues, such as *Hang Time Perc.*

Subsequently, we analyze variations in the general forecasting accuracy of TSF models across different metric classes. We use the Kruskal–Wallis test and find very high statistical evidence ($p < 0.001$, *2e-18*) that forecasting accuracy differs notably depending on the metric class. To gain deeper insights into these variations, we conduct post-hoc Mann–Whitney U-tests across all metric classes. The results, depicted in Figure 4.17a, demonstrate very high statistical significance ($p < 0.001$) for



(A) $RQ_{3.3}$. Heatmap of p-values obtained from the post-hoc Mann-Whitney U-test, illustrating the statistical significance of differences in MAE between pairs of metric classes.

(B) Distribution of MAE values grouped by metric class.

FIGURE 4.17: $RQ_{3.3}$. Cross-class comparison.

nearly all metric class pairs. These findings indicate significant differences in the MAE distributions between almost every pair of metric classes, with the sole exception being the comparison between *Waiting Times Perc.* and *Active Times Perc.* These differences are further illustrated in the violin plots presented in Figure 4.17b, clearly showing varied forecasting accuracy patterns depending on the metric type. For example, metrics such as *Hang Times Perc.* appear notably more challenging to predict compared to other classes. This observation is corroborated by the Mann–Whitney U-tests, which indicate highly significant differences ($p < 0.001$) in the MAE values when comparing *Hang Times Perc.* against all other metric classes.

Overall, our analysis underscores that the forecasting accuracy of TSF models strongly depends on the specific metric class being predicted. Metrics reflecting continuous software usage measurements, such as *Active Time Perc.*, *Waiting Time Perc.*, and particularly *Hang Time Perc.*, generally pose more complex predictive challenges compared to discrete-event metrics like *Crash Rate*. Indeed, crash rate metrics tend to display more predictable behavior, typically showing significant variability only during anomalous conditions. This can be also observed in Figure 4.17b, where MAE values of crash rate metrics are concentrated towards lower values, albeit exhibiting more pronounced outliers. Conversely, metrics associated with efficiency *and* unexpected issues, like *Hang Time Perc.*, emerge as the most challenging ones to accurately predict.

More in general, analyzing the connections between specific series properties and model effectiveness might shed light on strengths and weaknesses of each model. Figure 4.18 reports an illustrative example of an existing correlation between the model seasonal autocorrelation and the mean MAE of the forecasters we employed. This motivates future exploration of more advanced approaches to exploit such analysis and dynamically select the best model.

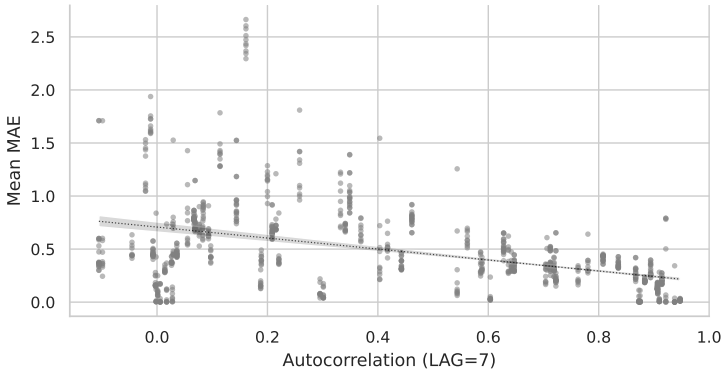


FIGURE 4.18: Relationship between Mean MAE of forecasting models and seasonal lag-7 autocorrelation (ACF). The black dotted line represents the linear regression fit between the mean MAE and the lag-7 ACF.

Answer to RQ$_{3.3}$: Foundation models consistently achieve the lowest median and mean MAE across most of software metric classes, significantly outperforming other models in two metric classes. However, we also observe that SARIMA yields superior performance on waiting times, though this improvement is only observed in the average rank. These results indicate that the effectiveness of each TSF models is not consistent across all types of metrics. These observations are further supported by our comparison of the general forecasting accuracy of TSF models, which highlights significantly different MAE distributions among metric classes, confirmed by very high statistical significance ($p < 0.001$).

RQ$_{3.4}$. To what extent does TSF accuracy change when extending the forecasting horizon?

Motivation. The goal of this research question is to assess how forecasting accuracy decreases when predicting longer-term software runtime metrics. The TSF models that we consider are able to generate *multi-step ahead* forecasts. This means that, at any given time, a TSF model can be queried to predict, for instance, the software runtime metrics of the next 14 days. It is expected that near-term predictions (*e.g.,* for the forthcoming days) will generally be more accurate than those for longer terms (*e.g.,* the following week). Through this research question, we aim to evaluate the extent to which the accuracy decreases as the forecasting horizon is progressively extended.

Methodology.

To achieve this research goal, we introduce the concept of *offset* (τ), which we define as the number of days between the last time step used as input and the *forecast target*. This essentially simulates scenarios where, at a given moment, the goal is to predict the metric for a specific future day. For example, an offset $\tau = 0$ indicates a scenario where the forecast target is the next day (*zero* denotes that there are no intermediate time steps), while an offset $\tau = 6$ corresponds to a scenario where the goal is to predict software runtime metrics for the same day in the following week. We choose to explicitly differentiate the *horizon* and *offset* concepts for sake of clarity during the reading; indeed, by horizon we denote the entire forecasting

period that we ask the models to generate, which is 14 in our setting, whereas the offset concerns how we traverse the horizon to compute the error for generating the i-th predicted future step. Thus, the offset is a concept that is related to our evaluation methodology instead of the experimental setup. Figure 4.4 graphically illustrates the concept of offset and forecast target in three representative scenarios. The first scenario represents the case where the goal is to predict the metrics of next day ($\tau = 0$), the second scenario targets predictions for the same day in the following week ($\tau = 6$), and the third scenario uses the same day of two weeks ahead as forecast target ($\tau = 13$).

For each specific offset h, we calculate the MAE of a given TSF model applied to a particular time series Y as follows:

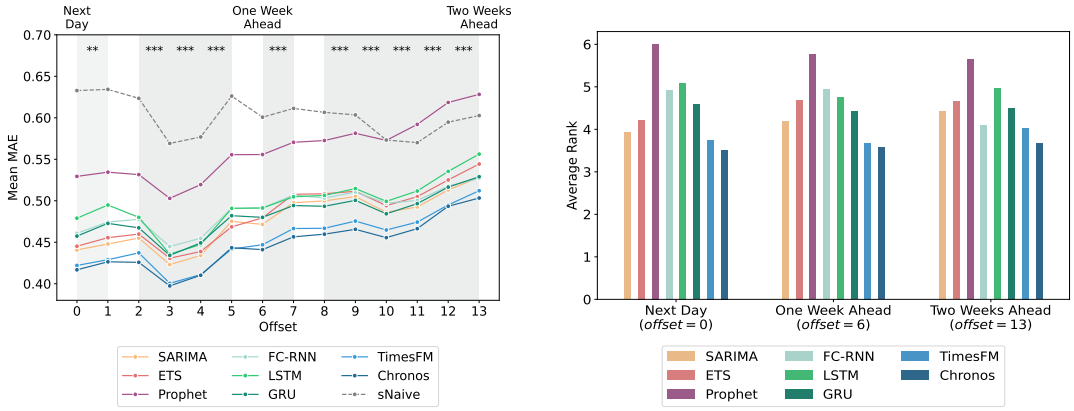
$$\text{MAE} = \frac{1}{n-k+1} \sum_{t=k}^{n} \left| F^t_{1+h} - Y_{t+h} \right| \tag{4.8}$$

where F^t_{1+h} represents the $(1+h)^{th}$ element of the forecast segment F^t, corresponding to the prediction for the $1+h$ steps ahead day (*i.e.,* the forecast target). Y_{t+h} indicates the actual measurement observed in the time series on that particular day. For example, with a $h = 6$ offset, MAE is calculated by considering exclusively the errors at the 7^{th} elements across all forecast segments of the time series. This corresponds to assessing the forecasting accuracy of a TSF method in predicting the runtime software metric for the same day in the following week.

As a result of this process, for each TSF method, we compute 1,540 MAE values, corresponding to each combination of time series and offset h, with $0 \leq h \leq 13$. We then summarize these outcomes by calculating the mean MAE for each model and offset h. Additionally, for each model, we outline the average rank across the three representative offset scenarios, namely next-day, one-week-ahead and two-weeks-ahead.

We conduct two statistical assessments for analyzing accuracy differences in long-term forecasting: (i) comparisons between each pair of consecutive offset values (*e.g.,* offset=3 *vs.* offset=4), and (ii) comparisons between each pair of models within a specific offset value (*e.g.,* one-week-ahead). The first assessment aims to evaluate the accuracy variation of TSF models when forecasting closer and farther future values. The second assessment analyzes the differences in models accuracy within a specific forecasting offset. By combining both assessments, we provide a detailed analysis of how the models accuracy changes as the forecasting horizon increases. For both assessments, we employ the Wilcoxon signed-rank test with Holm–Bonferroni correction for comparing the accuracy of models. In the first case, the offset values are treated as groups, while in the second, the forecasting models serve as the comparison groups.

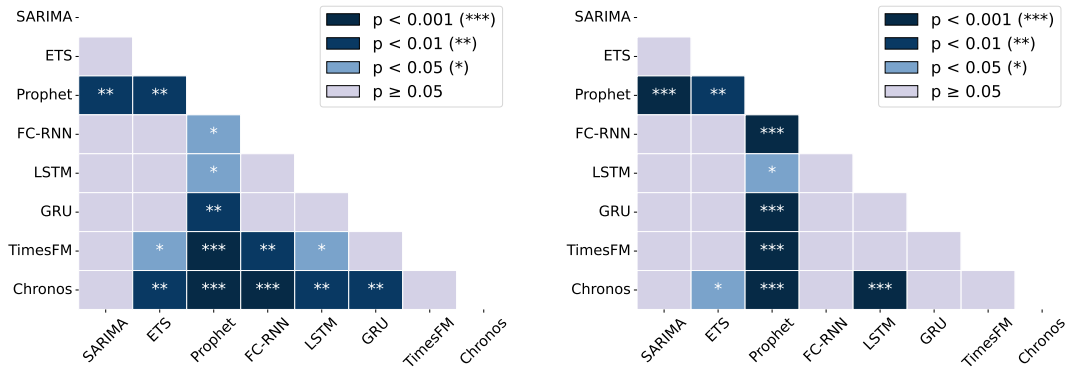
Results. Figure 4.19a reports, for each forecasting model, the mean MAE for each offset value. Under the assumption that farther values to predict correspond to greater uncertainty, as expected, higher offsets globally exhibit increased forecasting error values. This increasing trend is also observable by looking at each metric class individually. We include in the same figure the trend of sNaïve baseline for a practical reference. In terms of accuracy, foundation models show lower mean MAE across all the offset values, providing more reliable predictions across the entire forecasting horizon. By examining the figure, we notice a critical offset ($\tau \approx 10$) beyond which *Prophet* model methods begin to achieve worse forecasting



(A) Mean MAE. (B) Average Rank.

FIGURE 4.19: $RQ_{3.4}$. Relationship between forecasting accuracy and offset (h). The line plot includes annotations using asterisks (*) to denote the statistical significance at each h. Additionally, offset values with higher significance are highlighted with darker background shading. It's worth noting that mean MAEs and average ranks reported for the next-day predictions match the corresponding values reported in Figures 4.12b and 4.12a

.



(A) One Week Ahead. (B) Two Weeks Ahead.

FIGURE 4.20: $RQ_{3.3}$. Heatmap of p-values indicating the statistical significance of differences in MAE between TSF models across two prediction horizon offsets.

accuracy compared to the baseline. In other words, beyond that offset, its performance drops relative to the straightforward approach. The p-values obtained by comparing each pair of consecutive offset values are reported as annotations in Figure 4.19a using asterisks. We notice absent or lower significance among the first three offsets (0—2), while stronger significance is highlighted when increasing the future horizon. This suggests that the models accuracy difference, which in most cases corresponds to a degradation in performance, is more significant when predicting farther values ($\tau \geq 2$), despite a global decreasing trend in accuracy across all the forecasting horizons. Indeed, the first two offsets ($\tau < 2$) do not exhibit strong statistical significance. Figure 4.19b reports, for each model, the average rank specifically for next day, one-week-ahead and two-weeks-ahead representative scenarios. The figure clearly shows better values for foundation models in all the scenarios. Figure 4.20 illustrates the p-values obtained by comparing each pair of forecasting models in one-week-ahead and two-weeks-ahead scenarios. In Figure 4.20a, we observe statistical significance when comparing both the foundation models (particularly Chronos) against the majority of forecasting models, while Figure 4.20b depicts strong significance in the comparison between *Prophet* and the other models. This is indicating that the *Prophet* degradation in the last offset is more significant with respect to the other models. In addition, Chronos exhibits statistically significance when compared with ETS and LSTM in two-weeks ahead scenario.

Answer to $\mathbf{RQ}_{3.4}$: Foundation models exhibit superior forecasting accuracy over extended forecasting horizons compared to other models. However, statistical analysis shows that differences in MAE distributions become less significant as the forecasting horizon increases. This suggests that the gap in forecasting accuracy between models tends to narrow with longer prediction horizons.

4.3.3 Threats to validity

Construct validity The selection of TSF models represents a potential threat to construct validity of this study. The results obtained in performing the forecasting are directly related to the choice of TSF models to adopt. To mitigate this threat, we include in our selection several statistical approaches, recurrent neural networks and time series foundation models for covering a representative set of well-established and widely-adopted TSF methods. The presence of anomalies in the metrics also represents a potential threat during the evaluation. Indeed, the presence of pointwise outliers within the forecast segment could potentially lower the accuracy evaluation of the model even if the model is correctly predicting the behavior of the metric under non-abnormal conditions. However, when TSF is used for proactive decision-making (*e.g.,* resource allocation), being able to handle anomalies within the forecast segments is an advantage. Another threat concerns the choice of a metric for assessing the effectiveness of the TSF methods. The forecasting accuracy of the studied models was evaluated using MAE, as the time series were pre-standardized. However, it is important to note that other error metrics could partially alter the study outcomes. We only focused on software time series composed of daily data points. Our results may not generalize to other types of time aggregation granularity. Also, this obscures the intra-day variance and fluctuation dynamics, hence results may change if such dimensions are incorporated into the learning process. We leave this investigation as a future work. In addition, the windowing strategy adopted for training RNNs, including the chosen input and output lengths within

each window, may influence how the models exploit the historical context. As a result, the observed performance differences may depend on these design choices, and alternative configurations could lead to different RNNs behaviors. To mitigate this threat, we selected window parameters that balance a reasonable training set size with sufficient historical input (*i.e.,* two weeks), while respecting the architectural constraints of the models. Another potential threat of our work lies in the length of each time series contained in our dataset. Indeed, 110 time series with 400 observations each may be too small in some cases for effectively evaluating the ability of each forecasting method, especially given the wide spread of the amount of series in each class. Additionally, employing 360 measurements as historical context and 40 for evaluation may cause misleading results due to possible annual seasonality of the data.

Internal validity The implementation of RNN and SARIMA models in our study was carried out by using tensorflow and statsmodels, respectively. The foundation models have been imported and used with HuggingFace library. The choice of these specific libraries might introduce biases that could potentially influence the study results. Nonetheless, these are well-established libraries in the field of data analysis. The choice of fixed (non-tuned) hyper-parameters across all the series can significantly affect the forecasting accuracy of RNN methods. Results from hyper-parameter optimized RNN architectures might lead to different results. To mitigate this, we tried to maintain consistent hyper-parameter settings across different architectures. Specifically, all RNN-based methods were configured with an identical number of layers and units, while utilizing the same optimizer. Furthermore, we applied uniform early stopping strategies and epochs.

External validity The results of our empirical study may not generalize to other different software runtime metrics. Also, the data is limited to a single IT infrastructure, which may induce some specific fluctuations that do not generalize to other contexts. However, our analysis was based on a dataset comprising 110 different software runtime metrics collected from 29 distinct software applications, which span seven distinct metric classes. To the best of our knowledge, there are no other TSF studies that have utilized a dataset involving such diversity in software runtime metrics.

4.3.4 Conclusion

In this study, we analyzed the forecasting accuracy of various TSF models on software runtime metrics. Our results show that TSF models consistently outperform naïve baselines, demonstrating their suitability for predicting software runtime metrics. We found that time series foundation models deliver the highest forecasting accuracy on average, even when extending the forecasting horizon up to two weeks ahead. This result is particularly notable given that these models were employed in a zero-shot setting and were pre-trained on time series data unrelated to software metrics.

Although foundation models generally provide the best predictions, they do not consistently outperform other models across all individual time series. Indeed, we observed that even TSF models with comparatively lower average forecasting accuracy could still provide superior predictions for a significant number of time series. These findings highlight the absence of a universally optimal ("*silver bullet*") model that consistently surpasses others in every context, underscoring the necessity

of carefully selecting the appropriate TSF model to maximize forecasting accuracy. Additionally, our results indicate that forecasting accuracy substantially depends on the specific nature of the software metric under consideration. In fact, we observed clearly distinct patterns of forecasting accuracy when predicting different classes of metrics.

Overall, these findings encourage future research on understanding how the unique characteristics of each time series influence model performance, in order to guide the selection of the most effective forecasting model. For instance, one possible avenue could be the application of clustering approaches to better elicit what properties of the time series make the methods perform at their best (*e.g.,* autocorrelation in Figure 4.18). As future work, we also plan to investigate additional dimensions of time series forecasting in this context, including the incorporation of exogenous variables and multivariate time series as well as model enhancements such as hyperparameter tuning.

Chapter 5

Meta-Learning for Time Series-based Tasks in Software Engineering

Time series are extensively used for modeling a wide range of phenomena, spanning fields from climatology to economics and beyond [262]. Due to its versatility, the field of time series analysis has significantly grown over recent decades with the introduction of countless algorithms designed for tasks such as forecasting, classification, and anomaly detection [172], [263], [264]. These advancements have also started to gain attention within the Software Engineering (SE) domain [2], [4], [54], [135], [136], [265]. In particular, researchers have begun to employ time series analysis across a variety of SE tasks, often related to several critical software quality assurance activities. For example, time series forecasting has been employed to predict future software telemetry data [4], [54], thus supporting crucial tasks such as resource capacity planning [74], [75], [135], [136], software self-adaptation [29], [56] and more [53], [129], [130]. Time series anomaly detection has been utilized to automatically identify unexpected issues in online software systems and prevent major failures [5], [52], [55], [265]. Time series classification have been employed to address the inherent trade-off between result quality and execution time in software performance testing [2], [3], [102], [109].

Despite these developments, the effective application of time series analysis in SE remains fundamentally constrained by several critical challenges. One of the main difficulties lies in selecting the most appropriate algorithm for the target time series data. Indeed, the effectiveness of time series analysis algorithms often greatly varies depending on the characteristics of the data. This issue is particularly pronounced in SE contexts, where time series may represent diverse software metrics (*e.g.*, response times, software faults, CPU utilization) whose behavior is often highly variable [95] and influenced by external factors such as workload dynamics [131]. We observed clear evidence of this phenomenon in our previous empirical study [4], where we found that SARIMA [211]–a well-established algorithm for time series forecasting–performed suboptimally on most software telemetry data, yet proved particularly effective for certain specific types of software metrics. Figure 5.1 provides further evidence of this problem. The figure illustrates how frequently different algorithms achieve the top performance across six time series datasets spanning three distinct SE tasks: forecasting software telemetry data, anomaly detection in software systems, and steady-state detection in performance testing. Each pie chart shows the proportion of time series instances within a dataset for which each algorithm achieved the best performance. The figure clearly demonstrates that no single algorithm consistently achieves the best performance in more than 50% of the cases.

Consequently, even if one selects the algorithm that is on average the best for a specific SE task, it will still lead to suboptimal outcomes in more than half of the time series instances. This situation underscores the importance of carefully choosing the most suitable algorithm. However, this process may require specialized expertise and/or extensive experimentation, which may not be readily available to software engineers seeking to incorporate time series analysis into their SE processes.

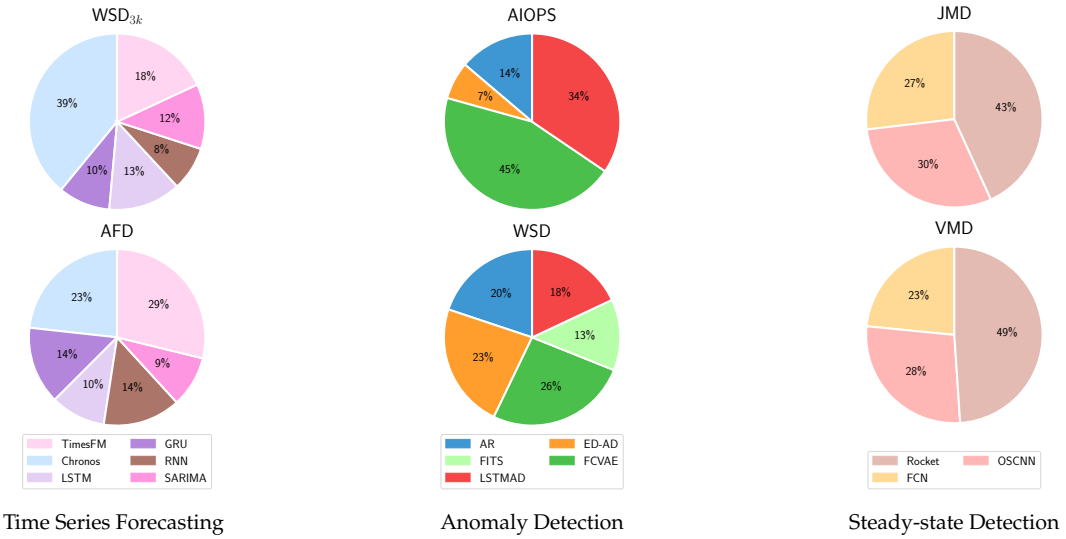


FIGURE 5.1: Distribution of best-performing algorithms on six software time series datasets (WSD$_{3k}$ [266], AFD [113], AIOPS [267], WSD [266], JMD [2], VMD [108]) spanning three different SE tasks (Time Series Forecasting, Anomaly Detection, Steady-state Detection). Each pie chart shows the percentage of instances in which each algorithm ranked first in performance compared to the other algorithms.

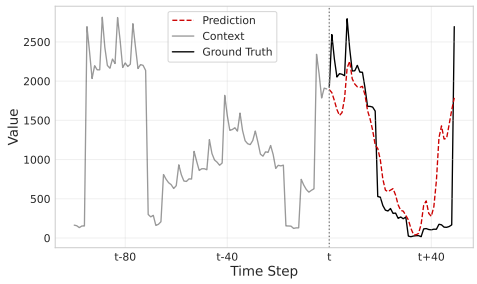
Over the years, the data science literature has proposed various approaches to automating the selection of algorithms for time series analysis. These approaches typically fall under the umbrella of meta-learning [140], leveraging descriptive features extracted from time series data to predict the most suitable algorithm. While such approaches have demonstrated effectiveness across several time series domains [137], [138], [139], [140], their efficacy within SE remains largely underexplored. Indeed, the application of meta-learning has the potential to provide substantial benefits across a wide range of SE activities. Yet, there is still little empirical evidence on whether–and to what extent–it can improve time-series-based SE tasks.

In this work, we aim to fill this knowledge gap by conducting the first empirical investigation on the effectiveness of meta-learning in time series–based SE tasks. To this end, we design MALIAS (MetA LearnIng for softwAre time-Series), a meta-learning framework that dynamically selects the best algorithm based on the characteristics of the time series and the SE task at hand. MALIAS first extracts about 700 features from each time series and employs these features to train a random forest model that predicts the performance of different algorithms. This process follows best practices in machine learning, such as dimensionality reduction and hyper-parameter tuning. The predicted algorithm performance are subsequently employed in the inference phase to automatically select and run the best algorithm for each given time series.

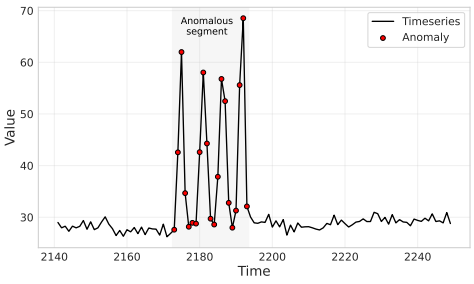
We evaluate MALIAS on 8,561 real-world software time series from six publicly available datasets covering three distinct time series analysis tasks (*i.e.*, two datasets for each task), namely: time series forecasting, anomaly detection, and classification. For forecasting, we selected datasets aimed at predicting future telemetry data of online software systems [113], [266]. For anomaly detection, we utilized datasets containing telemetry data from online software systems with labeled time steps marking known software issues [266], [267]. Lastly, for classification, we leveraged datasets addressing the steady-state detection problem [107], [108], which is commonly used to enhance the cost-effectiveness of software performance testing [2], [98], [102]. Results show that, when compared to the best performing algorithm for each dataset, MALIAS provides net improvements of +17.7*pp* and +9.5*pp* (SMAPE) for time series forecasting, +20.7*pp* and +3.1*pp* (AUC-PR) for anomaly detection, and +9.8*pp* and +19.8*pp* (F_β) for steady-state detection. Furthermore, MALIAS outperforms two well-known AutoML frameworks with very high statistical significance ($p < 0.0001$), while often providing comparable or even better training and testing times.

Our main contributions are as follows:

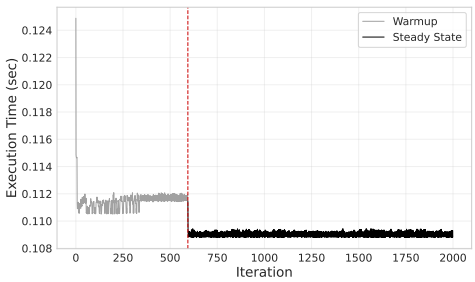
- We provide the first empirical evidence on the effectiveness of meta-learning for time series-based SE tasks.
- We present MALIAS a meta-learning framework for time series analysis in SE.
- We extensively evaluate MALIAS on six real-world publicly available time series datasets, spanning three different SE tasks.
- We make the source code, experimental scripts, and results of MALIAS publicly available [268] to support both developers and researchers.



(A) Time Series Forecasting of workload in a serverless environment [113].



(B) Time Series Anomaly Detection of software telemetry data from AIOps dataset [267].



(C) The steady-state phenomenon in the *fasta* benchmark [108].

FIGURE 5.2: Illustrative examples of practical scenarios involving time series-based tasks in software engineering.

5.1 Background and Related Work

5.1.1 Time Series-based Tasks in SE

The concept of time series has been widely employed in SE domain due to its flexibility in modeling software-related phenomena that evolve over time. Time series analysis tasks can be broadly divided in three main categories: *time series forecasting*, *anomaly detection*, and *time series classification*. Below, we report examples of application of time series analysis in SE for each of these three categories.

Time Series Forecasting (TSF)

TSF aims at predicting future values of a temporally ordered sequence by learning patterns and structures inherent to the historical observations, typically accounting for temporal dependencies, trends, seasonality, and noise within the data. TSF has been extensively applied in the SE domain, with methods ranging from ARIMA models [74], [129], [135] to neural networks [4], [53], [55]. A large body of research in this area focuses on forecasting telemetry data from online software systems—such as software performance metrics (*e.g.*, memory/CPU usage, response time) [4], [53], [54], [55] or system workloads [56], [74], [75]—to support tasks like capacity planning [74], [75], self-adaptation [29], and software rejuvenation [53].

Beyond software telemetry data, TSF has also been used to model the temporal evolution of software systems in terms of growth [227], [228], reliability [129], [130], bug report trends [135], and technical debt accumulation [136].

In this work, we focus on the prediction of software telemetry data as a case study to evaluate MALIAS, since forecasting telemetry data is one of the most common applications of TSF in SE. Figure 5.2a illustrates a real example of TSF applied to predicting the workload of an Azure serverless function [113]. The figure shows the number of function invocations over time at ten-minute intervals. A TSF model (Chronos [132]) is employed to generate predictions for the next 50 future values, namely from the time step "$t + 1$" to "$t + 50$". The grey line denotes the model's input context, the red dashed line indicates the forecasted values, and the ground truth is depicted in black.

Time Series Anomaly Detection (TSAD)

Anomaly detection involves identifying data points, patterns, or segments within a time-ordered sequence that significantly deviate from the expected behavior or underlying structure of the previously encountered data in the time series. Detecting anomalous data points within a set of observations is a critical capability in SE [55], [269]. This is particularly important in modern online software systems, where software undergoes continuous code changes [41] and is subject to highly variable workloads [131], making such systems more susceptible to unexpected issues [107], [111]. As a result, the adoption of anomaly detection techniques for software telemetry data has significantly increased in recent years, enabling the early identification of faults [5] and performance degradations [55]. Applications span over different domains such as spacecraft software [52], digital IT infrastructures [5], and microservices-based systems [270]. This growing interest has even led to the development of specialized TSAD benchmarks tailored for SE, such as TimeSeriesBench [265]. TimeSeriesBench employs a curated collection of software telemetry datasets with known anomalies, accompanied by associated algorithms and a standardized

process for evaluation and comparison. According to this benchmark, state-of-the-art approaches for TSAD in SE are predominantly based on deep learning models [265].

Figure 5.2b illustrates examples of known anomalies in software telemetry data from the AIOps dataset [267]. The anomalous data points are highlighted in red.

Time Series Classification (TSC)

TSC refers to categorizing entire sequences or segments of time series data into predefined classes based on learned temporal patterns and characteristics. A typical application of TSC in software engineering is the so-called *steady-state detection* problem in performance testing [107], [108].

This problem is particularly relevant in software that run on virtual machines that support Just-In-Time (JIT) compilation, such as Java, where performance measurements during the early iterations of test executions are often unstable due to runtime warm-up effects of JIT compilation. Such variability makes these early measurements unsuitable for rigorous performance evaluation.

Figure 5.2c illustrates this phenomenon using the *fasta* benchmark from the computer language benchmarks game [108], where the x-axis denotes successive benchmark executions and the y-axis indicates its execution time. In this example, the system reaches a steady state around the 600th iteration.

To detect the onset of steady-state behavior at runtime, researchers have developed a variety of techniques based on TSC. These approaches classify incoming segments of performance measurements as either "steady" or "non-steady," using either statistical heuristics [102], [109] or deep learning models [2], [3], and are typically employed to dynamically halt warm-up iterations at runtime during performance testing.

5.1.2 Time Series and Meta-Learning

Several previous studies have empirically demonstrated that there is no universally best algorithm for time series analysis [4], [172], [216], [232], [271]. As a result, a common practice is to rely either on the domain expertise of data analysts or on extensive empirical evaluation of available models to identify the algorithm that yields the best performance. However, expert knowledge is not always readily accessible, and exhaustive experimentation can be impractical due to time and cost constraints.

In light of these challenges, researchers have begun exploring automated approaches for selecting the best-performing models, leveraging the principles of meta-learning [137], [138], [140], [272], [273], [274], [275], [276], [277], [278], [279]. These approaches typically involve extracting descriptive features from time series data and training a predictor to identify the most suitable algorithm based on those features. For instance, Prudêncio *et al.* [137] introduced a method in which a meta-learner is trained to solve a classification problem, predicting the best algorithm for TSF. Shahoud *et al.* [138] proposed DSTMS, a descriptive schema for energy time series data, designed to support effective algorithm selection for energy forecasting tasks. Smith and Miles [140] formulated the algorithm selection task originally proposed by Rice [280] as a learning problem, demonstrating its applicability across various domains, including TSF. Talagala *et al.* [139] proposed the FFORMS framework, which classifies forecasting models based on over 30 time series features (*e.g.*, trend, seasonality, autocorrelation).

Building on the intuition of these works, this chapter introduces MALIAS, a meta-learning approach specifically tailored to the SE domain. To the best of our knowledge, none of the previous studies have explored meta-learning for time series-based tasks in SE.

5.2 Approach

We present MALIAS (MetA LearnIng for softwAre time-Series), a meta-learning framework designed to select and run the most effective algorithm for a given time series-based SE task.

5.2.1 Overview

MALIAS leverages a large number of time series features—which we refer to as *meta-features*—to select the most suitable algorithm for a given time series. To make this selection, MALIAS *learns* how each algorithm performs across individual time series based on their corresponding performance metrics, which we refer to as *target metrics*. These target metrics may vary depending on the task. For instance, they may be error-based in the case of TSF tasks (*e.g.,* SMAPE), or accuracy-based in the case of TSC tasks (*e.g.,* F_β score). At its core, MALIAS employs a *Random Forest* regressor model that, given the set of meta-features for a time series, predicts a vector of target metric values $\vec{m} = \langle m_1, m_2, \ldots, m_k \rangle$, where each m_i denotes the predicted target metric for candidate algorithm i on the provided time series. This prediction vector is then used to select the best-performing algorithm for the given SE task.

Figure 5.3 provides an overview of our approach, which consists of two stages: (i) an initial training stage and (ii) an application stage. The training stage begins with a set of training time series and a pool of k candidate algorithms. MALIAS executes each of the k algorithms on the training time series to obtain their respective target metrics. Simultaneously, it extracts *meta-features* from each time series. These meta-features, along with the corresponding metrics, are used to train a Random Forest model that learns to predict the vector of target metric values $\vec{m}$. This process yields a trained *meta-learner* capable of estimating algorithm performance based on the time series *meta-features*.

The application stage represents the practical usage scenario of MALIAS. Given a previously unseen time series, MALIAS first extracts the corresponding *meta-features* and then uses them as input to the meta-learner to predict the performance of the k candidate algorithms—namely, the vector of *target metrics* $\vec{m}$. Afterwards, MALIAS selects the best algorithm based on the predicted target metrics $\vec{m}$ and executes it on-the-fly. Depending on the specific SE task, the relevance of each *meta-feature* might change based on the relationship with the task *target metric*. For this reason, the proposed framework is task-dependent, namely a specific instance of MALIAS is uniquely dedicated to a specific SE task.

5.2.2 Meta-Features and Target Metrics

MALIAS collects hundreds of *meta-features* for each time series. Table 5.1 provides an illustrative example subset of the extracted features types. The actual features used in our work are specific instantiations of those types, created by selecting particular parameter values. For instance, a specific *fft_coefficient* feature can be defined by setting the *coefficient* parameter to `10` and the *attr* parameter to `abs`, thereby returning the real component of the complex value. As shown in the table, these features

TABLE 5.1: The table lists an illustrative subset of extracted feature types by name and description.

Feature type	Description
fft_coefficient	Coefficients of the one-dimensional discrete Fourier transform of the time series.
change_quantiles	Aggregated measure of changes in the time series within a corridor defined by lower and upper quantiles.
agg_linear_trend	Divide the time-series into equal chunks, compute an aggregate value for each chunk, then fit a linear regression to those chunk-averages against the chunk-index (0, 1, ...).
symmetry_looking	Boolean flag that indicates whether the distribution of the time-series "looks symmetric" by checking if the difference between its mean and median is small compared to its overall range.
large_standard_deviation	Boolean indicator of whether the standard deviation of the series is large relative to its range.
autocorrelation	Computes the correlation between the time series and its lagged (shifted) copy at a given lag.
energy_ratio_by_chunks	Ratio of the energy (sum of squares) contained in a chunk of the time series to the energy of the whole series, across several segments.
quantile	A selected quantile (e.g., 10th, 20th percentile) of the time series values, summarizing the distribution.
mean	The arithmetic average of the time series values, summarizing central tendency.
standard_deviation	The standard deviation of the time series values, summarizing dispersion around the mean.
	...

spread from the basic statistical properties (*e.g.*, quantile, autocorrelation) to more complex ones, such as frequency domain features [281]. The extraction process results in up to 771 distinct features[1] for each time series. Considering the substantial amount of extracted *meta-features*, we decide to apply a feature selection step for narrowing the selection to the most relevant *meta-features* for each specific task. Specifically, the feature selection technique is composed of two sequential filters. The initial filter aims to mitigate multicollinearity by removing redundant features that exhibit high similarity. In order to do that, the Pearson correlation coefficient is calculated among all the pairs of features, and those that have a correlation coefficient greater than *0.8* are ignored [282]. We refer to this step as *ρ-filter*. The second filtering step is performed using a *Recursive Feature Elimination* (RFE) [283] strategy[2], leveraging the feature importance score of the Random Forest model. The resulting features constitute the final set of *meta-features* that MALIAS will use to choose the best algorithm for the time series.

The *target metrics* represent the output of the Random Forest regressor integrated within MALIAS, namely the vector $\vec{m} = \langle m_1, m_2, \ldots, m_k \rangle$. MALIAS supports vectors of varying sizes depending on the number of k candidate algorithms available for the task. Additionally, the policy for recommending the best algorithm changes depending on the type of target metric (and corresponding task). For instance, in the case of TSF and error-based metrics, MALIAS selects the algorithm i that minimizes m_i. Conversely, for TSC/TSAD and accuracy-based metrics, MALIAS selects the algorithm i that maximizes m_i.

[1] https://tsfresh.readthedocs.io/en/latest/text/list_of_features.html

[2] We include the comparison of RFE with alternative feature selection strategies in the replication package [268], which supports the suitability of our choice.

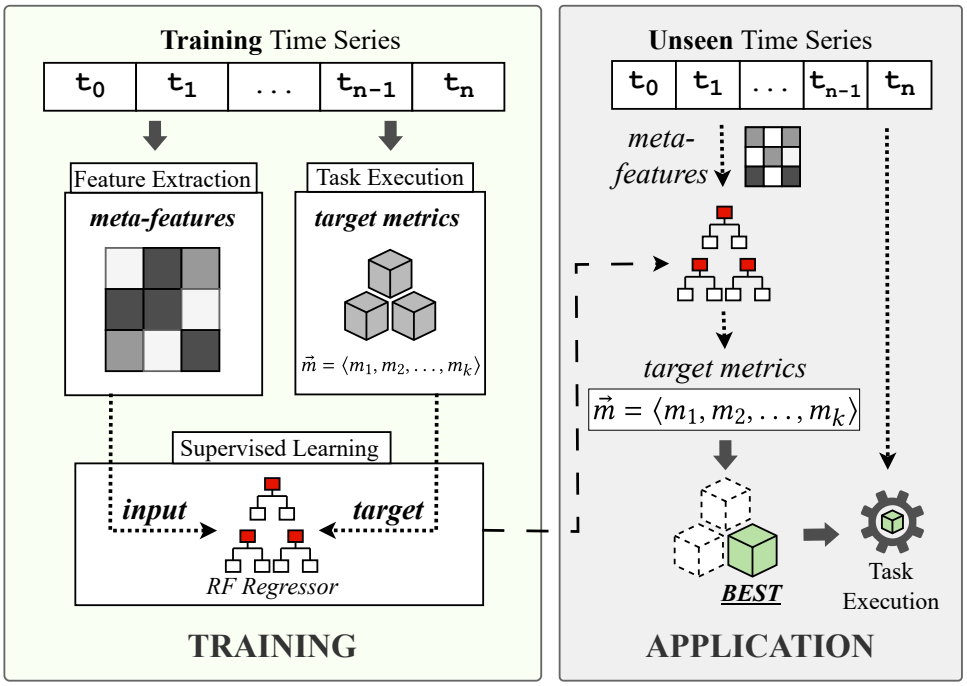


FIGURE 5.3: MALIAS Overview.

5.3 Time series based tasks

We evaluate MALIAS across three SE tasks, each representing a distinct type of time series analysis: time series forecasting (TSF), time series anomaly detection (TSAD), and time series classification (TSC). Specifically, MALIAS is assessed on (i) forecasting of software telemetry data (TSF), (ii) anomaly detection in online software systems (TSAD), and (iii) steady-state detection in software performance testing (SSD), which is framed as a TSC task.

5.3.1 Time Series Forecasting (TSF)

Datasets. For this task, we employ two publicly available, real-world datasets of software telemetry data: (i) *Azure Functions Dataset (AFD)* [113] and (ii) *Web Service Dataset (WSD)* [266]. AFD contains time series data representing the number of Azure Functions[3] invocations recorded for a 14-day period, aggregated at one-minute intervals. We initially preprocessed the AFD time series to aggregate the data into ten-minutes intervals, thereby reducing the length of each time series to 2,016 data points and consequently decreasing the computational cost of the experiments. Additionally, we focus on a subset of 288 functions exhibiting elevated workload variability, identified by examining the distribution of AFD time series values. Additional technical details are included in our replication package [268]. The second dataset (WSD) contains time series data that represent software telemetry data collected from three top-tier Internet companies providing large-scale web services (*e.g.*, Baidu, Sogou, and eBay). This dataset has been widely adopted in previous

[3] https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview.

time series-related SE studies [265], [284], [285]. The WSD dataset also includes synthetic time series; however, we chose to exclude them from our evaluation, as our goal is to assess the TSF capability of MALIAS solely on real-world software telemetry data. As a result of this filtering step, we obtain a total of 210 time series. Additionally, to mitigate the computational cost of our experiments, we truncate each time series to the first 3,000 values. For clarity, we refer to this reduced version of the dataset as WSD_{3k}.

Algorithms. We consider six candidate algorithms for TSF to evaluate MALIAS, covering the three most common classes of TSF models: classical statistical methods, recurrent neural networks (RNNs), and time series foundation models. As representative of classical statistical model, we select SARIMA [211], given its widespread popularity and adoption in recent studies [4], [286], [287]. From the RNN class, we include : Fully-Connected RNN (FC-RNN) [218], *Long Short-Term Memory* (LSTM) [221] and *Gated Recurrent Unit* (GRU) [219]. They are specifically designed to address time dependencies among input data, making them suitable for tasks involving sequential data. Moreover, recently introduced pretrained foundation models have demonstrated impressive effectiveness and flexibility, with the possibility to be used as zero-shot forecasters. From this category, we select two state-of-the-art time series foundation models: *TimesFM* [133] and *Chronos* [132]. Summarizing, our model selection for TSF considers six distinct models: SARIMA, FC-RNN, LSTM, GRU, TimesFM, and Chronos.

Evaluation. For this task, MALIAS is evaluated in performing multi-step-ahead univariate time series forecasting. We analyze the effectiveness of each forecasting model using the rolling origin evaluation technique [232], where the testing portion of each time series is partitioned into smaller overlapping sliding windows (*i.e.*, forecasting segments), and each model is subsequently tasked with generating predictions for all such segments. Each time series is initially partitioned into training, validation, and testing subsequences, using an 80-10-10 split strategy. The training set (TS_{train}) is used to fit the model, the validation set (TS_{val}) is used to tune the parameters of the SARIMA model (and is ignored for the other models), and the testing set (TS_{test}) is used to evaluate model performance—specifically, to obtain the *target metrics*. The prediction horizon is set to $H = 50$ future time steps. The length of the input window (ℓ) varies according the specific category of considered model [4]; the input window for RNN-based models is set to match the forecasting horizon ($\ell = 50$ in our setting), while the approaches that do not require any training, and instead perform zero-shot predictions (SARIMA, Chronos and TimesFM), use a maximum input sequence length of 512 time steps. Testing set windowing produces 102 50-length windows for AFD and 201 windows for WSD time series.

We evaluate each forecasting algorithm by adopting the Symmetrical Mean Absolute Percentage Error (SMAPE) [4], [232] on each testing set (TS_{test}). SMAPE values range from 0% to 200%, where 0% and 200% indicates perfect and worst forecasting accuracy, respectively. For each time series, SMAPE is calculated at each forecasting step h ($0 \leq h < H$), and the final score of each model in obtained by averaging over all H steps (mean SMAPE). Hence, this results in a mean SMAPE value for each time series and forecasting model, which together make up the *target metrics* of MALIAS.

5.3.2 Time Series Anomaly Detection (TSAD)

Datasets. We conduct TSDA experiments using two software time series datasets: (i) the *Web Service Dataset (WSD)*, which was also used in the TSF task, and (ii) *AIOPS* [267]. Both datasets consist of software time series with labeled data points annotated as anomalies. From the WSD dataset, we consider exclusively real-world software time series, excluding synthetic ones. We also exclude time series without anomalies, yielding a total of 161 WSD time series out of 510 initial time series. The resulting time series have a varying length ranging from 30,806 to 43,327 data points. The AIOPS dataset contains anomaly-labeled time series concerning software telemetry data, including a total of 29 time series collected from big companies (*e.g.,* Sougo, eBay, Baidu, and Tencent) each involving thousands of data points as well. The length of time series included in AIOPS dataset ranges from 17,568 to 299,053 data points. Both WSD and AIOPS datasets are widely employed in the TSAD SE literature [265], [288], [289]. Each time series of WSD and AIOPS is pre-partitioned into equal-sized 50-50% parts, one used for training (TS_{train}) and one for testing (TS_{test}).

Algorithms. We select five TSAD algorithms, each representing a distinct methodological family, as classified by Liu *et al.* [284]. We include LSTM-AD$_{\alpha}$ [290], a specific adaptation of LSTM-based model for performing the anomaly detection task on sequential data, and Autoregressive (AR) model [291], both categorized as *prediction-based* models. We also consider EncDec-AD [292], an LSTM-based encoder-decoder approach belonging to the *reconstruction-based* category, and the Frequency-enhanced Conditional Variational Autoencoder (FC-VAE) [293] model classified within the *VAE-based* family. Ultimately, we include FITS model [294], a lightweight 10k-parameters model for time series analysis, categorized as a *general-purpose time series* model.

Evaluation. The experimental setting of the TSAD task is based on the standard procedure proposed in the *TimeSeriesBench* [265] benchmark. The effectiveness of each TSAD algorithm is evaluated on each testing instance TS_{test} using the Area Under the Precision-Recall Curve (AUC-PR) [295], which serves as the *target metric* for this task. AUC-PR is widely adopted in TSAD tasks within SE [265], particularly because it effectively handles the typically imbalanced nature of such datasets. It captures the trade-off between precision and recall across varying classification thresholds. The AUC-PR value ranges from 0 to 1, with higher values indicating better performance in identifying positive class instances–*i.e.,* anomalies. To account for inherent detection latency in real-world applications, we adopt the refined version of the AUC-PR metric by applying a 3-step delay adjustment provided by the benchmark framework [265]. This adjustment is a latency-aware evaluation strategy that considers an anomaly to be successfully detected if it is identified within three time steps following its occurrence.

5.3.3 Steady-State Detection (SSD)

Datasets. For the SSD task, we leverage two datasets from prior related studies: the dataset introduced by Barrett et al. [108], referred to as Virtual Machines Dataset (VMD), and the dataset employed by Traini et al. [2], referred to as Java Microbenchmark Dataset (JMD). To the best of our knowledge, these constitute the most comprehensive publicly available datasets for steady-state detection. VMD includes 3,660 time series collected from 46 microbenchmarks encompassing 8 distinct JIT-equipped Virtual Machines. The microbenchmarks cover a variety of programming

languages, including Java, C, PHP and Python, and their executions are repeated over two different hardware configurations and two operating systems. Each time series is composed of 2,000 data points, and is annotated with the specific time step at which the steady-state is reached—marking the completion of the warm-up phase. We removed the time series extracted using PyPy, as the majority did not reach steady-state. JMD is the most extensive publicly available datasets of Java Microbenchmark Harness (JMH) performance measurements [57], containing measurements from 586 JMH microbenchmarks spread across 30 established open-source Java software systems. The dataset includes 10 time series for each microbenchmark, each one involving 3,000 consecutive microbenchmark iterations annotated with the specific iteration at which the steady state is attained.

We preprocess both VMD and JMD datasets to frame this task as a binary time series classification problem. Namely, each time series has been divided in smaller overlapping segments. Each segment is labeled as *unsteady* (negative class) if it contains at least one warm-up iteration; otherwise, it is marked as *steady* (positive class). In addition, we only included in the evaluation the time series that achieved steady state behavior, yielding a total number of 2,654 time series from VMD and 5,219 time series from JMD. To mitigate measurement redundancy across segments, we adopt a segmentation strategy with an adaptive step size [2]. Furthermore, we employ a customized resampling strategy designed to balance the number of steady and unsteady segments while maintaining a representative distribution across all microbenchmarks [2].

Algorithms. For the model selection, we reuse three state-of-the-art TSC algorithms that have been employed in a previous study for SSD [2], [3]: (i) Fully-Convolutional Network (FCN) [273], (ii) Omni-Scale Convolutional Neural Network (OSCNN) [179] and Rocket [176]. Both FCN and OSCNN are neural networks; the first one includes several stacked convolutional layers, while OSCNN is based on the Omni-Scale block introduced by Tang et al. [179]. Rocket combines convolutional kernels with a cross-validated Ridge Classifier [188] to perform the classification.

Evaluation. We perform the SSD experiments based on the procedure highlighted in prior work [2]. Specifically, the process combines an initial training phase, involving a supervised learning process over a set of training time series, and a subsequent inference stage over the testing set, employing a 5-fold cross-validation process. The splitting strategy accounts for data leakage prevention: for the JMD dataset, it ensures that the segments pertaining to the same benchmark are always located within the same fold, while for VMD, the time series related to the same VM are always located in the same fold. We evaluate the TSC algorithms using the F_β score, which serves as the *target metric* for SSD. We set the β value to 0.2 to prioritize precision over recall. This choice is motivated by the need to penalize more false positives than false negatives, as false positives have shown to have a more detrimental effect in the practical usage of SSD in performance testing [2].

5.4 Experimental Setup

In this section, we detail the experimental procedure used for evaluating MALIAS, along with the adopted metrics and implementation details.

5.4.1 Experimental Procedure

We evaluate the effectiveness of MALIAS on the three time series-based tasks introduced in the previous section. For each task, we use two datasets and configure MALIAS to include all candidate algorithms outlined in Section 5.3. For each dataset, the procedure involves a training stage followed by the application stage. In that, the time series within each dataset are partitioned into training and testing (*i.e.,* application) subsets. More specifically, the splitting strategy relies on a 5-fold cross-validation process, where each dataset is evenly partitioned into five disjoint subsets. In each iteration, four folds are used for training, while the remaining fold is used for testing the model. We iterate this process five times, ensuring that each fold is used exactly once for testing. Additionally, 20% of the time series are randomly selected within the four non-test folds to form a validation set. This validation set is employed for performing the feature selection and the hyperparameter tuning. Notably, the number of time series instances varies substantially across datasets—ranging from 29 (AIOPS) to 5,219 (JMD)— to allow evaluating the effectiveness of MALIAS under a wide range of data availability conditions.

Before starting the MALIAS training and evaluation process, we executed the k candidate algorithms for each of three tasks over their respective full datasets, to collect a k-length vector $\vec{m}$ of *target metrics* for each time series. Afterwards, we extract the *meta-features* for each time series instance.

In each cross-validation iteration, *target metrics* and *meta-features* from the training folds are used to train the Random Forest model, while the testing fold provides the data for MALIAS evaluation.

As described in Section 5.3, both TSF and TSAD experiments involve partitioning each time series instance into a training portion (TS_{train}) and a testing portion (TS_{test}). In line with the practical use of MALIAS, we extract the time series *meta-features* exclusively from TS_{train}. In contrast, the *target metrics* (SMAPE or AUC-PR) represents the algorithms' performance on TS_{test}.

For SSD tasks, when evaluating a time series, the extraction of the *meta-features* is performed on another time series belonging to the same microbenchmark. This experimental design choice aims to mirror a realistic usage scenario of SSD, where a microbenchmark is first executed once to gather its time series *meta-features* and, subsequently, MALIAS leverages these *meta-features* to automatically select the best algorithm for all future executions of that microbenchmark. In contrast, the *target metrics* (F_β scores) represent the performance of algorithms on the segments derived from the evaluated time series, as described in Section 5.3.

During the training stage, the supervised learning process of the Random Forest model (meta-learner) is structured as follows: for each time series, the *meta-features* serve as the input to the meta-learner, while the corresponding *target metrics* are normalized using a *min-max* strategy and concatenated to form the output vector $\vec{m}$, which serves as the target output of the meta-learner. In the application phase, the inference process follows a similar procedure: given a set of *meta-features* extracted from an unseen time series, the meta-learner predicts the corresponding vector $\vec{m}$ containing the *target metric* values for each algorithm. MALIAS then selects the algorithm with the most favorable predicted *target metric*. For TSAD and SSD, MALIAS selects the algorithm with the highest target metric values, as these correspond to accuracy-based scores (*i.e.,* AUC-PR and F_β). For TSF, MALIAS selects the algorithm with the lowest target metric values, since these correspond to error-based scores (*i.e.,* SMAPE).

5.4.2 Evaluation Metrics

We assess the effectiveness of MALIAS by focusing on two complementary aspects: (i) the frequency with which MALIAS yields improved or degraded performance compared to other time series analysis algorithms, by reporting the corresponding *Improvement Rates*, and (ii) the magnitude of performance deviation by comparing the *Target Metrics* and the *Oracle Performance Gap*. In the following sections, we provide a detailed explanation of each of these evaluation metrics.

Improvement Rates. The first measure concerns the frequency with which MALIAS achieves better performance than other time series analysis algorithms. In that, we compute the following evaluation metrics for each pairwise comparison:

- *Improvement (%)*: This measure represents the percentage of time series instances in which MALIAS outperforms the compared algorithm.
- *Regression (%)*: This measure represents the percentage of time series instances in which MALIAS provides worse performance than the compared algorithm.
- ***Net Improvement (pp)***: The arithmetic difference between the Improvement and the Regression percentages in percentage points (pp). This value reflects the net success frequency provided by MALIAS.

Target Metrics. We report summary statistics of the *target metric* values for our approach and other time series analysis algorithms. As detailed in Section 5.3, we use the SMAPE score for TSF, the AUC-PR values for TSAD, and the F_β score for SSD. Specifically, we present the mean values of these metrics using bar plots, along with the .95 confidence intervals (CIs). To assess whether the observed differences between MALIAS and the other algorithms are statistically significant, we apply the Wilcoxon signed-rank pairwise test with Holm-Bonferroni correction. Statistical significance levels are annotated on the bar plots using the following notation:
$p \leq 0.05$ (*), $p \leq 0.01$ (**), $p \leq 0.001$ (***) and $p \leq 0.0001$ (****).

Oracle Performance Gap (OPG). We also assess the relative performance gap of MALIAS compared to an oracle-based approach [296], which always selects the optimal algorithm for each time series instance. The concept of an oracle is commonly used in the evaluation of dynamic ensemble selection methods [297], [298]. If MALIAS perfectly selects the best algorithm for each time series instance, then it will match the oracle performance. It is worth noting that the oracle performance represents an upper bound for MALIAS in the case of TSAD and SSD tasks, and a lower bound in the case of TSF. We refer to this measure as the OPG score, which quantifies the relative performance loss with respect to the oracle. Let S denote the performance achieved by MALIAS (*i.e.*, mean SMAPE for TSF, mean AUC-PR for TSAD, and mean F_β for SSD), and let S^* represent the ideal performance of the oracle. We define the OPG score as: $\frac{|S^* - S|}{S^*} \times 100$. This measure (where lower values are better) reflects how closely MALIAS approaches the optimal performance achievable within the given experimental setup.

5.4.3 Implementation Details

We rely on established ML libraries—such as *TensorFlow* and *HuggingFace*—to implement TSF algorithms. The TSAD experiments are implemented based on the *EasyTSAD* suite [265], which offers a configurable experimental framework encompassing all the considered anomaly detection algorithms and evaluation metrics. The

SSD experiments are implemented reusing the data extraction, training and testing scripts provided by previous related work [2], [3], [107], utilizing the same experimental setting and parameters. Concerning the MALIAS implementation, the feature extraction module is based on *TsFresh*[4], resulting in up to 771 features for each time series. The random forest regressor, along with the feature selection and hyper-parameter tuning phases, are implemented using the *scikit-learn* library. Further implementation details are available in the replication package [268].

The total amount of time required for conducting the entire experimental process—including the algorithms execution and MALIAS evaluation—is approximately one week. The complete experimentation was performed on a a Linux server equipped with 40 Intel(R) Xeon(R) 2.30GHz CPUs and 78Gb of RAM.

5.5 Results

In this section, we present our research questions, describe the methodology used to address them, and discuss the results referring to the fourth research contribution (RC_4). The results of the study presented in this chapter are organized around five research questions, each addressed in the following subsections.

5.5.1 $RQ_{4.1}$: To what extent does MALIAS enhance forecasting of software telemetry data?

Objective. The goal of this request question is to assess if and how effectively MALIAS can improve TSF performance when applied to software telemetry data.

Methodology. We execute MALIAS following the cross-validation strategy outlined in Section 5.4.1. Specifically, at each iteration, we first perform feature selection and hyperparameter tuning using the training and validation folds. We then evaluate the predictions generated by MALIAS on the test fold using SMAPE. The feature selection procedure results in 39 *meta-features* for the AFD dataset and 20 for the WSD_{3k} dataset. After obtaining the final set of SMAPE values, we compute: (i) the *Improvement Rates* of MALIAS relative to the six considered TSF algorithms—SARIMA, FC-RNN, LSTM, GRU, TimesFM, and Chronos; (ii) the statistical significance of the differences in the *Target Metrics* using the Wilcoxon signed-rank test; and (iii) the *Oracle Performance Gap (OPC)*.

Results. Table 5.2 shows the improvement rates of our approach relative to baseline TSF algorithms. The results clearly demonstrate the advantage of adopting MALIAS for TSF of software telemetry data. MALIAS improves SMAPE across a significant portion of the evaluated time series instances, with improvements observed in 40.3% to 81.6% of instances for AFD, and 14.3% to 79% of instances for WSD_{3k}. However, it also leads to regressions in up to 33% of instances for AFD and up to 25.7% of instances for WSD_{3k}. Nevertheless, MALIAS ultimately achieves a substantial net improvement over all baseline algorithms, ranging from +17.7 percentage points (*pp*) to +63.9 *pp* in AFD, and from +9.5 *pp* to +60.5 *pp* in WSD_{3k}.

Figure 5.4 reports the mean SMAPE of MALIAS and TSF algorithms, along with their 0.95 confidence intervals. We observe that MALIAS achieves the lowest mean errors on both datasets, reporting a mean SMAPE of 37.89% for AFD and 6.78% for WSD_{3k}. The results of the Wilcoxon signed-rank test highlight a very high statistical significance in the differences between the SMAPE values produced by MALIAS

[4] https://tsfresh.readthedocs.io/en/latest/.*Accessed on July 4, 2025.*

TABLE 5.2: Improvement rates of MALIAS over baseline TSF algorithms

	Azure Functions Dataset (AFD)		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs. Chronos*	40.3%	22.6%	**+17.7**
MALIAS *vs. TimesFM*	44.4%	23.6%	**+20.8**
MALIAS *vs. GRU*	51.4%	28.8%	**+22.6**
MALIAS *vs. RNN*	60.1%	31.6%	**+28.5**
MALIAS *vs. LSTM*	64.9%	33.0%	**+31.9**
MALIAS *vs. SARIMA*	81.6%	17.7%	**+63.9**

	Web Service Dataset (WSD_{3k})		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs. Chronos*	14.3%	4.8%	**+9.5**
MALIAS *vs. TimesFM*	68.6%	25.7%	**+42.9**
MALIAS *vs. GRU*	78.1%	18.6%	**+59.5**
MALIAS *vs. RNN*	79.0%	18.6%	**+60.5**
MALIAS *vs. LSTM*	76.7%	18.1%	**+58.6**
MALIAS *vs. SARIMA*	78.6%	19.0%	**+59.5**

TABLE 5.3: MALIAS selection rate in TSF datasets.

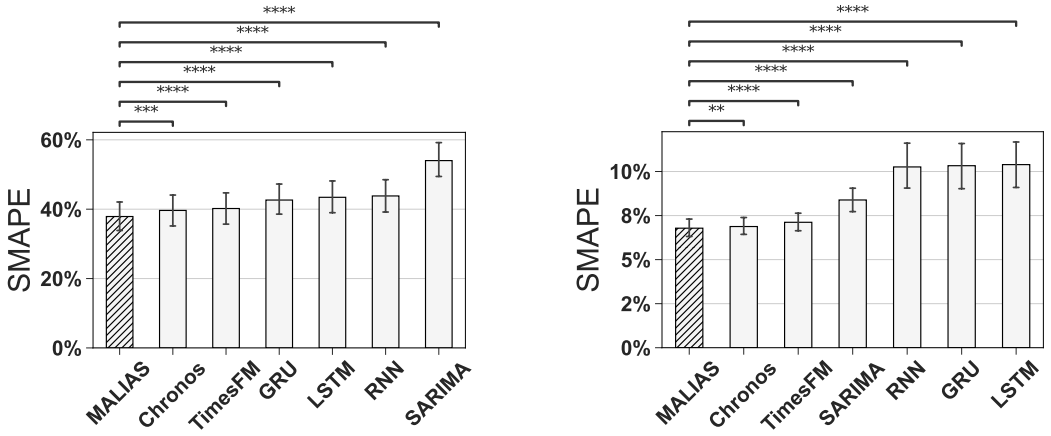
Dataset	#TS	Selection Rate (%)					
		Chronos	*TimesFM*	*GRU*	*RNN*	*LSTM*	*SARIMA*
AFD	288	**37.2** %	<u>31.9</u> %	19.8 %	8.3 %	2.1 %	0.7 %
WSD_{3k}	210	**81.0** %	<u>5.7</u> %	3.3 %	2.4 %	5.2 %	2.4 %

The highest is reported in **bold**, the second-highest is <u>underlined</u>.

and those by other TSF algorithms, with $p < 0.001$ across all MALIAS-baseline comparisons. This result indicates a consistent improvement over all baseline TSF algorithms on both datasets.

To better understand these improvements, we analyze the algorithm selections made by MALIAS. Table 5.3 presents the selection rates of algorithms chosen by MALIAS across both TSF datasets. We observe that the distribution of selection rates differs significantly between the two datasets. For the AFD dataset, the selections are more diverse: Chronos is the most frequently selected model (37.2%), followed by TimesFM (31.9%) and GRU (19.8%). In contrast, the WSD_{3k} dataset exhibits a more skewed selection pattern, with Chronos being selected 81% of the times. All other models are chosen less than 10% of the times. Interestingly, even though MALIAS predominantly selects Chronos, it still achieves an overall improvement of +9.5 *pp* over Chronos (as shown in Table 5.2).

We also observe an alignment between the algorithm selection rates by MALIAS and the distribution of best-performing models depicted in Figure 5.1. For instance, the top-3 best-performing TSF models on the AFD dataset match the three most frequently selected algorithms by MALIAS. Similarly, for WSD_{3k}, the best-performing model, Chronos, is also the one most frequently selected by MALIAS. However, we also notice a significant mismatch between these percentages. For example, MALIAS selects Chronos 81% of the time, even though it is the best-performing model in just 39% of cases. This discrepancy potentially explains the OPC values of 5.05% for WSD_{3k} and 16.86% for AFD shown in Table 5.4. These results highlight that MALIAS has not reached oracle-level performance, suggesting there remains room for improvement.



(A) Azure Functions Dataset (AFD).

(B) Web Service Dataset (WSD_{3k}).

FIGURE 5.4: Mean SMAPE for MALIAS and TSF algorithms; annotations indicate Wilcoxon signed-rank test results.

TABLE 5.4: OPG of MALIAS across tasks.

Task	**Target Metric**	*MALIAS OPG (%)*	
TSF	SMAPE	*AFD* 16.86 %	*WSD_{3k}* 5.05 %
TSAD	AUC-PR	*AIOPS* 5.15 %	*WSD* 7.86 %
SSD	F_β	*VMD* 2.63 %	*JMD* 3.24 %

Answer to $RQ_{4.1}$: MALIAS consistently improves the performance of TSF algorithms, with very high statistical significance ($p < 0.001$). MALIAS yields a net improvements of +17.7 *pp* and +9.5 *pp* when compared to the best-performing TSF baseline.

5.5.2 $RQ_{4.2}$: To what extent does MALIAS enhance anomaly detection in software systems?

Objective. This research question investigates the benefit of MALIAS in performing TSAD in online software systems.

Methodology. The evaluation process follows the same approach employed in $RQ_{4.1}$—*i.e.*, using cross-validation. The feature selection process leads to 9 *meta-features* for AIOPS, and 112 for WSD dataset. The anomaly detection capabilities of MALIAS are evaluated using AUC-PR. The evaluation is carried out by analyzing the improvements rates of MALIAS over five baseline TSAD algorithms—LSTM-AD_α, AR, EncDec-AD, FC-VAE, and FITS—along with the mean AUC-PR values and OPG scores.

Results. Improvement rates for TSAD are presented in Table 5.5. We observe that the percentages of improvements are always higher than regressions. Specifically, for the AIOPS dataset, MALIAS delivers a net improvement ranging from +20.7 *pp* (*vs.* FCVAE) to +93.1 *pp* (*vs.* FITS). The percentages of regressions, which range from 3.4% to 27.6%, are indeed considerably lower than those of improvements, which

TABLE 5.5: Improvement rates of MALIAS over baseline TSAD algorithms

	AIOPS Dataset		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs.* $LSTMAD_{\alpha}$	34.5%	3.4%	**+31.0**
MALIAS *vs. FCVAE*	48.3%	27.6%	**+20.7**
MALIAS *vs. AR*	75.9%	10.3%	**+65.5**
MALIAS *vs. EncDecAD*	86.2%	13.8%	**+72.4**
MALIAS *vs. FITS*	96.6%	3.4%	**+93.1**

	Web Service Dataset (WSD)		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs.* $LSTMAD_{\alpha}$	23.0%	19.9%	**+3.1**
MALIAS *vs. FCVAE*	41.0%	31.7%	**+9.3**
MALIAS *vs. AR*	57.1%	25.5%	**+31.7**
MALIAS *vs. EncDecAD*	45.3%	34.8%	**+10.6**
MALIAS *vs. FITS*	59.6%	28.0%	**+31.7**

range from 34.5% to 96.6%. In the WSD dataset, we observe improvements ranging from 23% to 59% and regressions from 19.9% to 34.8%. As a result, MALIAS achieves net improvements between +3.1 *pp* and +31.7 *pp* on WSD instances.

TABLE 5.6: MALIAS selection rate in TSAD datasets.

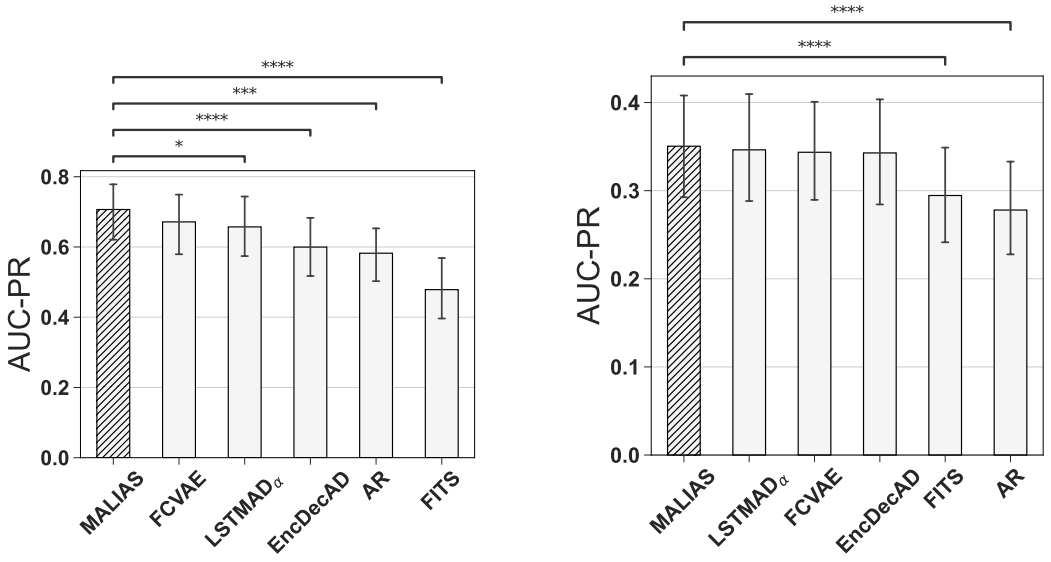
Dataset	#TS	Selection Rate (%)				
		LSTM-AD$_{\alpha}$	*FCVAE*	*AR*	*EncDecAD*	*FITS*
AIOPS	29	**62.1%**	<u>24.1%</u>	13.8%	0%	0%
WSD	161	**52.8%**	<u>22.4%</u>	11.8%	8.7%	4.3%

The highest is reported in **bold**, the second-highest is <u>underlined</u>.

As shown in Table 5.6, the model most frequently selected by MALIAS is LSTM-AD_{α}, although the FCVAE model is also chosen in over 20% of the instances in both datasets. Notably, this task involves the smallest number of time series—190 in total. Despite the limited size of the training and validation datasets, MALIAS consistently produced effective algorithm recommendations, outperforming all baseline TSAD models.

Figure 5.5 presents the mean AUC-PR scores for MALIAS and the baseline TSAD algorithms. On the AIOPS dataset, MALIAS achieves the highest mean AUC-PR score (0.71). Among the baseline algorithms, the most accurate for AIOPS is FCVAE, with a mean AUC-PR of 0.67. MALIAS outperforms all the TSAD baselines with statistical significance ($p < 0.05$), with the sole exception of FCVAE. On the WSD dataset, MALIAS also achieves the highest AUC-PR (0.350), although the performance differences with the baselines are less pronounced. For example, the best-performing TSAD baseline, LSTM-AD_{α}, achieves an AUC-PR of 0.346. Statistically significant improvements are observed only over FITS and AR ($p < 0.0001$). Table 5.4 presents the OPG score achieved by MALIAS, showing a gap of 5.15% in AIOPS instances and a gap of 7.86% for WSD.

Answer to $RQ_{4.2}$: MALIAS consistently outperforms all TSAD baseline models, delivering a net improvement of +20.7 *pp* for AIOPS and +3.1 *pp* for WSD against the best-performing baseline.



(A) AIOPS Dataset (B) Web Service Dataset (WSD)

FIGURE 5.5: Mean AUC-PR for MALIAS and TSAD algorithms; annotations indicate Wilcoxon signed-rank test results.

5.5.3 $RQ_{4.3}$: To what extent does MALIAS enhance steady state detection in software performance testing?

TABLE 5.7: Improvement rates of MALIAS over baseline SSD algorithms.

	Virtual Machines Dataset (VMD)		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs. Rocket*	24.9%	15.0%	**+9.8**
MALIAS *vs. OSCNN*	49.5%	19.3%	**+30.2**
MALIAS *vs. FCN*	61.3%	19.5%	**+41.8**

	Java Microbenchmark Dataset (JMD)		
MALIAS *vs.* Model	Improvement (%)	Regression (%)	***Net Improvement (pp)***
MALIAS *vs. Rocket*	39.4%	19.6%	**+19.8**
MALIAS *vs. OSCNN*	33.2%	12.6%	**+20.6**
MALIAS *vs. FCN*	59.2%	31.2%	**+28.0**

Objective. This research question aims to assess the effectiveness of MALIAS for SSD in the context of software performance testing.

Methodology. We evaluate MALIAS for SSD similarly as we did for TSF ($RQ_{4.1}$) and TSAD ($RQ_{4.2}$). Specifically, we follow the cross-validation experimental procedure presented in Section 5.4, where the task performance is evaluated using F_β score. The feature selection step eventually yields 8 *meta-features* for JMD and 5 for VMD. We compare the performance of MALIAS with that of three SSD algorithms, namely Rocket, FCN and OSCNN. The final comparison is made by considering the improvement rates, mean F_β scores, and the OPG scores.

Results. Table 5.7 shows the improvement rates achieved by MALIAS compared to each baseline SSD algorithm. In both the VMD and JMD scenarios, MALIAS consistently delivers higher improvement percentages than regressions. Specifically,

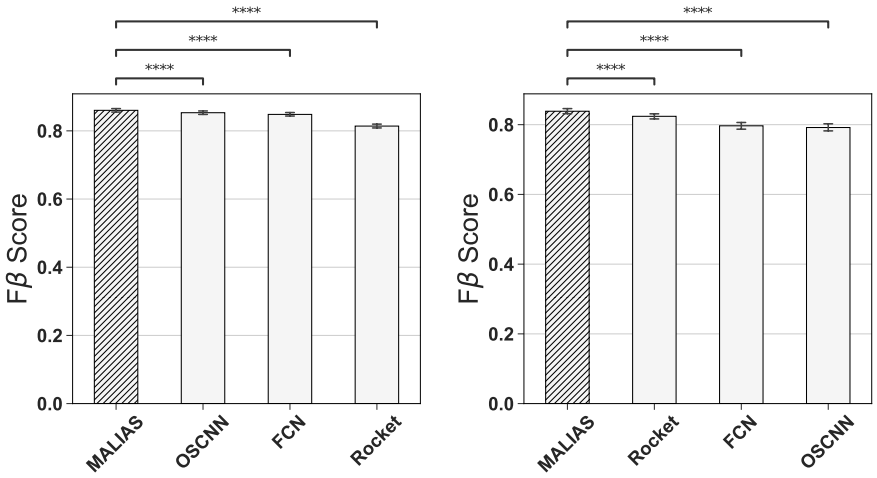
TABLE 5.8: MALIAS selection rate in SSD datasets.

Dataset	#TS	Selection Rate (%) *Rocket*	*OSCNN*	*FCN*
JMD	5,219	<u>40.1</u> %	**53.7** %	6.2 %
VMD	2,654	**58.1** %	<u>26.9</u> %	15.0 %

The highest is reported in **bold**, the second-highest is <u>underlined</u>.

for the VMD dataset, improvements range from 24.9% to 61.3% of the time series instances, whereas regressions are observed in only 15% to 19.5% of instances. This results in net improvements ranging from +9.8 *pp* to +41.8 *pp*. Similarly, for the JMD dataset, improvements span from 33.2% to 59.2%, with net improvements between +19.8 *pp* and +28 *pp*. These findings highlight the consistent effectiveness of MALIAS for the SSD task.

Figure 5.6 reports the mean F_β score for each approach in JMD and VMD datasets. We notice that MALIAS always achieves the best mean F_β score (0.860 for JMD and 0.839 for VMD) and that it outperforms all the other baseline SSD algorithms with very high statistical significance ($p < 0.0001$). For Table 5.4, we also find that MALIAS nearly approaches oracle-level performance with OPG of only 2.63% for VMD and 3.23% for JMD.



(A) Java Microbenchmark Dataset (JMD) (B) Virtual Machines Dataset (VMD)

FIGURE 5.6: Mean F_β scores for MALIAS and SSD algorithms; annotations indicate Wilcoxon signed-rank test results.

Concerning the selection rates reported in Table 5.8, we observe that Rocket is the most frequently selected model for the VMD dataset (58.1%). This aligns well with the results shown in Figure 5.1, where Rocket emerges as the best-performing SSD algorithm in 49% of the time series instances. Interestingly, for the JMD dataset, we notice a different trend: OSCNN becomes the most frequently chosen model by MALIAS (53.7%), despite Rocket remaining the most frequent best-performing model on JMD, as indicated in Figure 5.1. This discrepancy can be attributed to the learning process being guided by the target metric values rather than by success frequency. Nonetheless, even in the JMD dataset, MALIAS still provides a substantial net improvement over Rocket (+19.8 *pp*), as well as better F_β scores, with very high statistical significance ($p < 0.0001$).

Answer to RQ$_{4.3}$: MALIAS outperforms all baseline SSD algorithms with very high statistical significance ($p < 0.0001$), delivering net improvements of +9.8 *pp* (VMD) and +19.8 *pp* (JMD) when compared to the best-performing baseline.

5.5.4 RQ$_{4.4}$: To what extent does MALIAS generalize across datasets of different domains?

Objective. In this research question, we investigate the extent to which MALIAS generalizes across datasets from different domains. Specifically, we aim to understand whether MALIAS can be effectively employed in a zero-shot setting when pretrained on datasets originating from other domains.

Methodology. To answer this research question, we train MALIAS on the *meta-features* and *target metrics* of one dataset and evaluate its performance on another dataset related to the same task. Specifically, as our experimental setup includes two datasets per task, we first train MALIAS on one dataset and subsequently test it on the other. For instance, in the TSF task, we train MALIAS on the AFD dataset and evaluate it on WSD$_{3k}$, and vice-versa. We follow the same procedure for all three time series-based tasks: TSF, TSAD, and SSD. We refer to this variant of our approach as MALIAS $_{gen}$.

After obtaining the SMAPE values (for TSF), AUC-PR values (for TSAD), and F$_\beta$ scores (for SSD) of MALIAS $_{gen}$, we compute the *net improvements* compared to two baselines: (i) a standard instance of MALIAS (i.e., trained on the training set from the same dataset), and (ii) the best-performing baseline algorithm for the specific task and dataset. As in earlier research questions, net improvement is defined as the percentage difference between instances where MALIAS $_{gen}$ outperforms the baseline and those where it underperforms.

TABLE 5.9: Net improvements of MALIAS $_{gen}$.

Task	Testing Dataset	Training Dataset	*MALIAS $_{gen}$ vs. best-performing model (pp)*	*MALIAS $_{gen}$ vs. MALIAS (pp)*
TSF	WSD$_{3k}$	AFD	-6.2 (*vs. Chronos*)	-15.7
TSF	AFD	WSD$_{3k}$	-0.7 (*vs. Chronos*)	-18.1
TSAD	AIOPS	WSD	-10.3 (*vs. FCVAE*)	-44.8
TSAD	WSD	AIOPS	-6.8 (*vs. LSTMAD$_\alpha$*)	-8.1
SSD	JMD	VMD	+0.8 (*vs. Rocket*)	-19.6
SSD	VMD	JMD	-3.8 (*vs. Rocket*)	-13.8

Results. Table 5.9 reports the net improvements of MALIAS $_{gen}$ compared to both a standard MALIAS instance and the best-performing baseline algorithm. When compared to the standard MALIAS instance, we observe that MALIAS $_{gen}$ consistently yields negative net improvements. This indicates that the proportion of time series instances where MALIAS $_{gen}$ improves MALIAS performance is lower than those where it worsens it. Specifically, net improvements range from -8.1 to -44.8 *pp*. This suggests that the relationship between *meta-features* and *target metrics* is closely tied to the characteristics of a specific dataset. Compared againsts the best-performing baseline algorithm, we observe slightly better net improvements, though they remain predominantly negative. The only exception is the JMD dataset, where MALIAS $_{gen}$ yields a marginal net improvement of +0.8 *pp*. In all other cases, MALIAS $_{gen}$ results in negative net improvements ranging from -0.7 to -10.3 *pp*.

Answer to $RQ_{4.4}$: The prediction capabilities of MALIAS do not generalize across datasets from different domains. MALIAS $_{gen}$ exhibits lower performance compared to both the standard MALIAS instance and the best-performing baseline.

5.5.5 $RQ_{4.5}$: How does MALIAS perform relative to automated machine learning (AutoML) frameworks?

Objective. In the preceding research questions, we demonstrated that MALIAS can automatically select a suitable algorithm for a given target time series, providing substantial benefits over the use of any individual algorithm. Here, we aim to assess how MALIAS performs in comparison with other similar approaches that automatically select and integrate algorithmic predictions. Specifically, we seek to compare its effectiveness and computational efficiency with those of established AutoML frameworks.

Methodology. To define a fair and representative baseline for comparison, we selected widely-used AutoML frameworks from the open-source ecosystem. The selection of AutoML frameworks was performed through a systematic exploration of GitHub repositories, by querying with the keyword "*AutoML*" and limiting the results to the top 20 most-starred projects with more than 500 forks. Frameworks were included if actively maintained (*i.e.,* not archived) and natively provide utilities and algorithms for dealing with time series data. Following this screening process, two relevant open-source AutoML frameworks were identified as baselines: (i) *AutoGluon*[5] and (ii) *FLAML*[6]. AutoGluon uses models bagging and stack ensembling, leveraging the predictions of each individual algorithm as extra features in the fitting stage, and, at prediction time, it averages the predictions of all these individual models to generate the final prediction scores. FLAML selects a set of candidate learning algorithms and defines a corresponding hyperparameter search space for each. It then performs training and validation across these candidates and selects the model achieving the best evaluation score for prediction. Although FLAML supports stacked ensembling with multiple estimators, this option is disabled in its default configuration. Based on the target task, the AutoML frameworks select the appropriate data preprocessing and validation strategies, such as stratified sampling for classification and temporal splitting for forecasting.

We compare MALIAS with both AutoGluon and FLAML using one representative dataset per task. Specifically, we employed the Azure Functions Dataset (AFD) for TSF, the AIOPS dataset for TSAD and Virtual Machines Dataset (VMD) for SSD. As in the previous research questions, we report, for each approach, the improvement rates along with the distribution of the metric scores for each task: SMAPE for TSF, AUC-PR for TSAD, and F_β for SSD. Ultimately, we compare and analyze the time spent by each approach in carrying out the training and testing phases for each time series.

Experimental Configuration. Given the extensive configurability of AutoGluon and FLAML, we preserved their default settings to approximate a realistic *out-of-the-box* usage scenario. At the same time, the experimental setup must ensure a meaningful and balanced comparison between MALIAS and the selected AutoML frameworks, which inherently support a broader range of algorithms and more powerful optimization capabilities. AutoGluon includes a set of validated pre-defined

[5] https://auto.gluon.ai/stable/index.html
[6] https://microsoft.github.io/FLAML/

configurations—termed *presets*—designed to achieve varying performance levels; we employed the `medium_quality` preset to ensure a model pool comparable to that of MALIAS. Under this setting, the framework exploited eight models for TSF and eleven models for TSAD and SSD, excluding the stacked ensemble. As the *presets* parameter inherently controls the computational effort, no explicit time budget was imposed on the training process for this framework. AutoGluon natively supports the TSF task through the `TimeSeriesPredictor` class, whereas the TSAD and SSD tasks were executed using the `TabularPredictor` class with the task type set to `binary`. FLAML does not support a *presets*-like parameter, so we handled the computational cost and the search space using the time budget, defined as the maximum amount of seconds granted for executing the selection process over a single series. For each task, the FLAML training time budget was set to the average training time spent by MALIAS, computed per time series. FLAML automatically adjusts its model selection and hyperparameter optimization strategies according to the specified time budget. For the TSF task, FLAML was configured with the `ts_forecast` task type to learn from past data and predict the next time horizon. Given the use of rolling-origin cross-validation, predicting multiple testing windows required refitting FLAML on the same time series, which aligns with how the framework handles sequential forecasts. However, performing multiple fittings from scratch would create an unfair comparison with the other approaches, that do not require such repetition. Therefore, we adopt a minimal configuration for this refitting process, restricting its time budget to one second and applying refitting solely to the best estimator selected by FLAML during the training phase. Additionally, we disable FLAML's internal validation during this stage. The TSAD and SSD tasks are performed setting the `classification` parameter as task type, using the time series window data points as individual features.

The evaluation metric used for algorithm selection was automatically determined by each AutoML framework according to the task type.

Execution Times. The training procedure of MALIAS requires executing all candidate algorithms over the *training pool* of time series, plus the fitting process of the selected algorithm for the *target* time series. To shed light on its efficiency within the context of meta-learning, we compare its training and testing time with those of the selected AutoML frameworks. In that, we report the average training and testing times per time series for each approach and dataset. For each k-fold iteration, the training cost of MALIAS is normalized by the number of time series within the corresponding testing fold, yielding the unit cost associated with a single series.

We computed the training time of MALIAS (T^M_{train}) as:

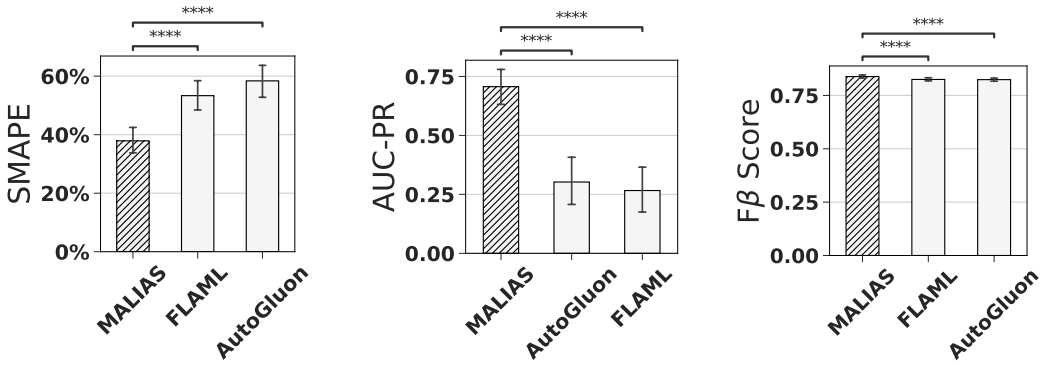
$$T^M_{\text{train}} = \frac{T_{\text{ext}_F} + T_{\text{M-fit}_F}}{k_{test}} + T_{\text{fit}_A}$$

where k_{test} is the number of time series included in the testing fold of the dataset, T_{ext_F} indicates the average time required to extract all the features and target metrics across the training folds, $T_{\text{M-fit}_F}$ refers to the total time spent for the feature selection, hyper-parameter tuning and the training of the meta-learner embedded in MALIAS, while T_{fit_A} is the average time required for training the algorithm selected by MALIAS when applied to a specific time series. The T_{ext_F} and $T_{\text{M-fit}_F}$ are computed in average over the five cross-validation iterations.

The testing time of MALIAS (T^M_{test}) is defined as the average time required by the chosen algorithms for generating the predictions when applied to a single time series, following the evaluation process described in Section 5.3. Namely, for TSF, the

testing time corresponds to the duration required for generating the testing windows for each time series (*i.e.*, 102 50-length windows in the AFD dataset); for TSAD, it corresponds to the time needed to compute anomaly scores for the testing portion if the time series (50% of the AIOPS series data points); and for SSD, it is defined as the time required to classify segments of length 100 for each microbenchmark execution (fork) in the VMD dataset.

AutoGluon and FLAML, instead, were applied independently to each time series, with training and testing times computed as the average across all series.



(A) TSF - AFD (B) TSAD - AIOps (C) SSD - VMD

FIGURE 5.7: Mean scores for MALIAS and AutoML frameworks; annotations indicate Wilcoxon signed-rank test results.

TABLE 5.10: Comparison between MALIAS and AutoML frameworks.

MALIAS *vs.*	Improvement (%)	Regression (%)	***Net Improvement (pp)***
	TSF - Azure Functions Dataset (AFD)		
AutoGluon	86.8%	13.2%	**+73.6**
FLAML	83%	17%	**+66**
	TSAD - AIOPS Dataset		
AutoGluon	100%	0%	**+100**
FLAML	96.6%	3.4%	**+93.1**
	SSD - Virtual Machines Dataset (VMD)		
AutoGluon	51.1%	34.2%	**+16.8**
FLAML	47%	35.7%	**+11.3**

Results. Table 5.10 reports the improvement rates achieved by MALIAS in comparison with the AutoML baselines. When performig the TSF task, MALIAS improved by +73.6*pp* over AutoGluon and +66 pp over FLAML. In the TSAD task, MALIAS reported all wins over the 29 timeseries in the AIOPS dataset compared with AutoGluons (+100*pp*), and a net improvement of +93.1*pp* compared with FLAML. In the SSD task, the net improvements achieved by MALIAS are smaller but consistently positive, with gains of +16.8*pp* compared with AutoGluon and +11.3*pp* with FLAML on the VMD dataset. Notably, the SSD experiment reveals that the number of regressions—defined as the time series where MALIAS underperforms compared with the baselines—exceeds 30% of the dataset, a substantially higher rate with respect to the other tasks. Overall, MALIAS outperformed both AutoML baselines across all the three tasks. Figure 5.7 presents bar plots with the distribution of performance metrics for each task. In TSF and TSAD tasks, it is evident that the baseline

scores are considerably lower than the ones obtained with MALIAS. However, in the SSD task, all the three approaches achieved a mean F_β score over 80%, suggesting similar performance. Nonetheless, as annotated in the same figure, the Wilcoxon test returned very high statistical significance ($p < 0.0001$) among all the comparisons.

The principal differences between MALIAS and the AutoML baselines lie in their training procedures and algorithm selection strategies. In this setting, MALIAS is trained over a pool of time series drawn from the same domain on which it is subsequently evaluated (*i.e.*, same dataset), whereas the AutoML baselines select the optimal model solely based on the time series under analysis. Additionally, MALIAS uses time series characteristics and metric values for carrying out the selection, while the AutoML frameworks primarily rely on multiple algorithm predictions for this purpose. The findings suggest that utilizing cross-series information within the same domain is an effective approach to guide algorithm selection, and that this process can be effectively driven in light of the characteristics of the target series.

TABLE 5.11: Comparison of Training and Testing Times Between MALIAS and AutoML.

Task	Training time			Testing time		
	MALIAS	**AutoGluon**	**FLAML**	**MALIAS**	**AutoGluon**	**FLAML**
TSF	**158**	391.42	200.58	11.65	**4.16**	29.26
TSAD	1,733	**720.76**	1,791.11	39.49	2.11	**1.81**
SSD	34	**1.19**	38.07	**0.05**	0.27	0.09

Times are reported in seconds, averaged per time series.

Table 5.11 reports training and testing times for MALIAS, AutoGluon and FLAML. In the TSF task, the results indicate that MALIAS attains a faster training time, averaging 158 seconds per time series, corresponding to a 59% decrease relative to AutoGluon (391 s) and a 21% decrease relative to FLAML (201 s). In carrying out the TSAD, AutoGluon achieved the fastest training time, with an average of 721 seconds per time series. In percentages, this reduces by 58% the MALIAS training time (1,733 s) and by 60% the FLAML training time (1,791 s). For SSD, AutoGluon remains the fastest option with an average training time per time series of 1.19 seconds, decreasing by 95% the time required by MALIAS (34 s) and by 97% the training time required by FLAML (38 s). Overall, FLAML reports training times very close to those of MALIAS, likely due to the time budget configuration, which was derived from MALIAS average training duration. Table 5.11 also reports the testing times for each. Across the evaluated tasks, all the approaches exhibit heterogeneous testing performance. In the TSF task, it records a runtime of 12 seconds, higher than AutoGluon (4 s) but considerably lower than FLAML (29 s). For TSAD, MALIAS required 39 seconds per time series, which is longer than both baselines; however, in the SSD task, it achieved the lowest inference time (0.05 s), outperforming both AutoGluon (0.27 s) and FLAML (0.09 s). These results suggest that the efficiency of each approach is task-dependent, and that the method runtime behavior is influenced by the characteristics of the underlying dataset or task configuration, rather than by a consistent computational advantage. In summary, during the training stages MALIAS proved to be the most efficient for the TSF task, while AutoGluon reported significantly lower times in TSAD and SSD scenarios. In the testing phase, each approach appeared to be the most efficient in at least one specific task. However, the net improvements and scores indicate that the actual effectiveness of MALIAS is significantly higher than that of the baseline methods.

Answer to $RQ_{4.5}$: In terms of effectiveness, MALIAS consistently outperforms both AutoML baselines (AutoGluon and FLAML) across all the tasks, with results demonstrating very high statistical significance ($p < 0.0001$). However, regarding training and testing times, its efficiency varies across tasks, achieving the fastest execution only in the *TSF training* and *SSD testing* settings.

5.6 Practical Implications

In this study, we empirically demonstrate the effectiveness of meta-learning in supporting time series analysis for software engineering tasks. We emphasize that all datasets used in this work come from real-world systems operating under natural conditions, ensuring that our findings reflect authentic behavior rather than artifacts of simulation or synthetic data. This section explores how the proposed approach can be practically integrated into software engineering processes.

Time series forecasting is widely adopted for monitoring real-world applications during post-deployment [4], [55]. Based on our results, meta-learning proves highly beneficial for enhancing the forecast quality of software telemetry data. To operate, MALIAS requires a set of training time series to fit the meta-learner. In a real-world scenarios, where software systems are instrumented through Application Performance Monitoring (APM) tools, obtaining such data is straightforward due to the large number of software components that are continuously monitored [5], [30]. As a practical example, the training set can be extracted from time series capturing the runtime behavior of software components from the same system infrastructure, or those belonging to same metric classes [4]. We envision that the application of meta-learning can provide substantial benefits in software engineering activities that involve forecasting software telemetry data. For instance, more accurate forecasting may lead to better-informed capacity planning [75] and more effective self-adaptation in software systems [29].

MALIAS has also demonstrated strong effectiveness in time series anomaly detection, where its capability to dynamically select the most appropriate algorithm has enhanced the identification of software anomalies. Similar to the TSF scenario, the large volume of metrics collected by APM tools during the monitoring process can be leveraged in practical settings to establish consistent relationships between specific time series characteristics and the effectiveness of each detection model. Consequently, embedding meta-learning within these contexts could drive significant progress in AIOps and software monitoring.

Prior work in software performance testing has demonstrated that the use of deep learning models can support the detection of the steady state of performance, thus enabling a reasonable balance between result quality and testing time [2], [3]. However, even in such cases, selecting the optimal model remains a challenging task. As shown in Figure 5.1, different models may offer varying levels of accuracy depending on the characteristics of the performance measurements. Our experimental results indicate that meta-learning can enhance the accuracy of steady-state detection, suggesting its potential for deployment in practical software performance testing scenarios. For instance, practitioners can use measurements collected from microbenchmarks executed on previous versions of the software under test as a set of training time series to train the meta-learner. This training process can be performed once or periodically updated to mitigate information drift in performance

measurements caused by codebase evolution. During practical use of the framework, the practitioner should first run the microbenchmark to obtain a representative time series from which features can be extracted. These features can then be fed to the meta-learner to enable the selection of the most suitable algorithm, which will subsequently be integrated at runtime during performance testing to halt warm-up iterations dynamically. As such, meta-learning can help achieve a better trade-off between the quality of test results and testing time, which has been an enduring challenge in performance testing [98].

5.7 Threats To Validity

Construct Validity. A primary threat concerns the choice of *meta-features* and time series analysis algorithms employed in the study. We mitigate this threat by using large number of time series features (771) and by considering algorithms of various kinds. The choice of the target metrics also affects the outcomes achieved in this work. However, our selection is grounded in the practical usage scenario of each task and is aligned with common literature practices [2], [232], [265].

Internal Validity. The implementation of each candidate algorithm introduces potential limitations and affects the accuracies reported in this study. We aimed to mitigate this threat by relying on widely employed ML libraries and by adhering the implementation guidelines and tools established in prior work [2], [4], [265].

External Validity. The findings reported in this chapter may not generalize to time series extracted from other domains or employed for other tasks, as they might have different characteristics and the impact of the meta-feature might change accordingly. However, we employed six dataset of real-world time series spreading three main categories of time series-based SE tasks. Additionally, MALIAS has been evaluated using univariate time series, and its capacity to model high-dimensional coupled features has yet to be established.

5.8 Conclusion

This study provides the first empirical evidence supporting the effectiveness of meta-learning in time series–based SE tasks. We introduce MALIAS, a meta-learning framework evaluated across three distinct SE tasks using six publicly available datasets of real-world software time series. Results show that MALIAS consistently outperforms both the best-performing models and two well-established AutoML frameworks across all considered tasks. This improvement can be attributed to MALIAS ability to automatically identify the most suitable time series analysis algorithm given the specific characteristics of the data. The implementation of MALIAS is publicly available, facilitating reproducibility and enabling future research and practitioners to reuse and adapt our approach.

Despite the significant improvements delivered by MALIAS, our study highlighted challenges in generalizing its predictive capability across datasets from different domains. Specifically, our findings indicate that MALIAS cannot effectively function as a pre-trained approach in zero-shot settings and requires domain-specific training for optimal performance. This limitation opens opportunities for future research to explore meta-learning techniques tailored explicitly for transfer learning in time-series-based SE tasks. We encourage further investigation into approaches

that can address these generalization challenge, which will drive our future research agenda.

Chapter 6

AI-Assisted Regressions Analysis of User Digital Experience in an Industrial Context

Modern enterprises heavily rely on digital devices, such as desktop computers, to carry out their business operations. Therefore, a dependable IT infrastructure is essential to ensure seamless business continuity. However, avoiding significant software problems that can hinder business activities presents an ongoing challenge for companies. Due to the current fast-to-market trend [41], modern software applications are continuously updated to meet new requirements and to experiment new features, and the time reserved for quality assurance activities during software development is often limited [33], [41]. In that, each software update poses a potential source of issues that can directly impact business digital devices. For instance, recent studies suggest that seemingly non-invasive software changes, such as maintenance activities, can unexpectedly compromise software quality [42], [43].

To deal with this context, companies often rely on Application Performance Monitoring (APM) tools [30] to monitor the execution behavior of software applications (*e.g.*, Dynatrace[1], Datadog[2], Aternity[3]). These tools continuously record metrics related to different aspects of software execution in the field (*e.g.*, fault rates, logging, execution times), and enable the adoption of quick remediation actions when relevant software failures occur [30]. Despite the utility of these tools, the effective exploitation of collected metrics for diagnosis and resolution of relevant software issues remains challenging. Software metrics are known to be subject to severe fluctuations (*e.g.*, due to varying workloads [131] and/or execution environments [57], [95]), and it is often difficult to determine to what extent these fluctuations are "normal" or symptoms of actual anomalies. Furthermore, anomalies are not uncommon in practice, and it can be challenging to assess the significance of an anomaly, thus to determine which ones should undergo root cause analysis.

In this chapter, we present *RADig-X* (Regression Analysis of User Digital Experience), a tool to support the identification and analysis of digital experience issues, and we report on our experience in the adoption of this tool in a large real-world company, namely Micron, which monitors more than 30,000 digital devices around the world. *RADig-X* leverages a combination of AI algorithms and a ranking heuristic to: (i) determine actual anomalies in runtime metrics gathered from APM tools, (ii) prioritize the relevance of anomalies according to their impact on the overall IT infrastructure, and (iii) rank problematic software updates that are correlated with

[1]Dynatrace. https://www.dynatrace.com
[2]Datadog. https://www.datadoghq.com
[3]Aternity. https://www.aternity.com

relevant anomalies. Through an empirical evaluation using real-world monitoring data, we demonstrate that *RADig-X* achieves a 94% improvement in F1-score for anomaly detection compared to the current practice employed within the company. Furthermore, our results suggest that *RADig-X* has potential in identifying software updates that exhibit a notable correlation with software application crashes. *RADig-X* is currently deployed within Micron to support the diagnosis and problem resolution of digital experience issues. To date, *RADig-X* has facilitated the detection of multiple regressions, including two specific ones that had substantial impacts on the digital user experience. Since the adoption of *RADig-X*, there has been a 14.66% enhancement in the index measuring the quality of digital experiences within the company.

The rest of the chapter is structured as follows. Section 6.1 presents the industry context of this work. Section 6.2 describes our formative study and the main challenges that we aim to tackle. Section 6.3 outlines the main components of *RADig-X*. In Section 6.4, we present our research questions along with experimental design and results. Section 6.5 discusses the insights from the usage in practice of *RADig-X*. Section 6.6 presents the threats to validity of our study, Section 6.7 presents the related work, and Section 6.8 concludes this chapter. The content of this chapter is derived from the corresponding conference paper [5]. Consequently, references to the first author and all co-authors in this chapter follow the author list as it appears in the publication.

6.1 Context of this work

Micron is a leader company in memory solutions manufacturing. As of 2023, it is the 4th largest semiconductor company in the world, with a presence in 17 countries and a workforce of ∼49,000 employees.

Micron employees make an extensive use of digital devices (*e.g.*, desktop computers) to carry out their business activities. However, these devices are subject to frequent software updates that make them susceptible to unexpected performance degradations and/or software failures. These software issues can have a severe impact on employee productivity, such as causing a slowdown in business activities and potential economic effects for the company.

To mitigate the impact of these issues, Micron closely monitors more than 30.000 business devices using a popular APM tool[4]. A dedicated monitoring team is in charge of promptly identifying software issues that can have a significant impact on employee productivity. Each of the 30,000+ devices is equipped with a *APM tool* agent that records a variety of runtime metrics, such as response time of user actions, boot times, software crashes and resource usage, thus totaling to several gigabytes of data per day. These metrics can then be accessed either directly by the monitoring team through visual dashboards, or programmatically through REST APIs to observe the state of each device. Due to the vast volume and variety of metrics collected by APM tool, it is often difficult to gather a comprehensive picture of the health status of the company IT infrastructure. To this aim, Micron relies on the APM tool's *IDX* that summarizes, in a unique metric, the observed quality (the higher *IDX* the better quality) of the employee's digital experience. *IDX* provides two key benefits: (i) it offers a unique metric that acts as a proxy measure for the health status of the entire IT infrastructure, and (ii) it allows for comparison with

[4]For privacy reasons, we name as *APM tool* the one used here, and as IDX the adopted digital experience quality index.

IDX of other similar companies through a dedicated dashboard, thereby providing useful reference baselines. *IDX* can be configured to suit the specific business needs of a company, for example by assigning larger weight to the runtime metric of a particular software application. *IDX* calculation considers as input a subset of runtime metrics collected by the APM tool agent on the devices; based on these, a local *IDX* value (*IDX*-subscore) devoted to each precise runtime metric considered is calculated. These local *IDX* values are subsequently combined according to the configuration established by the company to arrive at a unique final *IDX* score (global *IDX* score). However, it can be challenging to understand how fluctuations in a specific runtime metric (*e.g.,* number of crashes in a particular software application) will ultimately affect *IDX*. Indeed, the process of deriving the *IDX* sub-scores starting from the raw metrics is often obscure for the company, since its computation also considers the metrics associated to other APM tool customers, which might not be available to the company. In addition, *IDX* has a fixed refresh rate, and therefore it is not directly accessible in real-time by the company. This can be a problem when, for example, a (buggy) software update is deployed on a significant number of devices, and the monitoring team needs to promptly detect any *IDX* drops to prevent negative effects on employee operability.

The monitoring team continuously monitors both raw metrics and *IDX* to spot anomalies through APM tool dashboards. Nonetheless, the frequency of software updates on employee devices poses a persistent threat that can prevent employee operability at any given moment. Software updates in Micron devices can be triggered directly by the employees or forced by the company. The latter are typically scheduled upfront and deployed on a large number of devices to maintain enterprise software up-to-date across all the company devices (*e.g.,* due to security reasons). In such cases, potentially problematic software updates can be deployed over a large amount of devices, with significant consequences on the entire IT infrastructure. To prevent updates from causing failures on systems, the mass release phase is usually preceded by the test deployment phase, where a subset of devices is randomly chosen to assess the potential impact of the updates. Despite these mitigation strategies, Micron often struggles to avoid side-effects due to software updates, especially due to the scale of the problem. About 70% of devices contain more than a thousand installed applications, and software change events per month are in the order of millions, including installations, upgrades and removals.

Figure 6.1 presents the digital experience monitoring process adopted in Micron. *APM tool* agents deployed on employee devices continuously record in a data storage software/system runtime metrics, and software changes (*i.e.,* software installations, updates and removals). Then, the monitoring team accesses the recorded data mainly through *APM tool* dashboards, which provide a collection of facilities to query and visualize software runtime metrics and software changes data. In addition, the monitoring team has developed a series of custom applications, on top of the APM tool APIs, to enable automated queries and analyses, such as automating metric checks to trigger an alert if the number of software crashes increases over a threshold.

The monitoring team is in charge of three main activities within Micron: (i) *problem identification* , (ii) *root cause analysis*, and (iii) *issue reporting*. *Problem identification* entails understanding if there are significant anomalies in the monitored runtime metrics that can negatively impact employee productivity. *Root cause analysis* involves extracting the potential causes of a manifested runtime metric anomaly (*e.g.,* buggy software updates or installations), while *issue reporting* consists in reporting the identified anomalies and the associated possible causes to the team in charge of

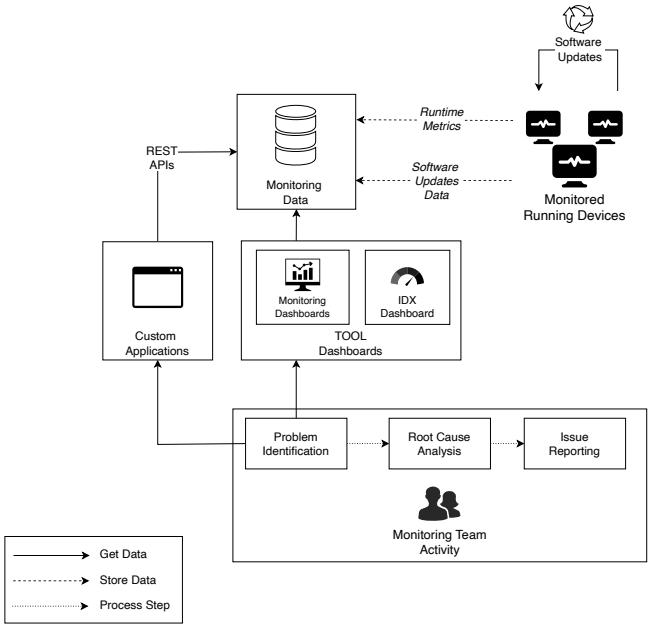


FIGURE 6.1: Workflow of Digital Experience Monitoring Process

managing the entire IT infrastructure.

6.2 Formative study

We conducted an initial formative study to understand the challenges faced by the monitoring team. The first author spent the first two months of the study on-site, together with the monitoring team, to directly experience the activities and challenges faced by the team. In the first two weeks, daily training meetings were held on the main context-related concerns, such as: team organization, used monitoring tools, IT infrastructure and support applications. Afterwards, the first author actively participated in the daily activities carried out by the monitoring team, such as problems identification of ongoing *IDX* regressions and root cause analysis, for a total training period of two months. At the end of this phase, a discussion with all the co-authors has been held in order to extract the main concerns.

We distilled three main challenges: (i) *anomaly detection*, (ii) *digital experience impact evaluation* and (iii) *root cause analysis*.

Anomaly detection. The monitoring activity of devices involves ongoing tracking of application quality metrics, *e.g.*, application crashes and wait times, which results in time series of metric values concerning applications that are spread across a wide range of devices. Digital experience regressions occur when the system behavior is changing unexpectedly. Hence, some of metric time series may present anomalies that differentiate the running state from the past usual patterns. In that, a major challenge consists in detecting anomalies within metric time series, by distinguishing between seasonal fluctuations in recorded metrics and true anomalies. Examples of seasonal fluctuations are due to device running hours. For instance, less application crashes can be due to many idle or powered-off devices that are not running any application (*e.g.*, for different working hours in different world time zones). These kinds of fluctuations do not impact employees user experience, so they should not be reported.

Current internal practices mostly rely on human-driven analysis, which can be expensive given the large number of metrics that affect *IDX*.

Anomaly impact evaluation. Industry processes lead to frequently evolving changes to infrastructure and software that can bring anomalous situations due to bugs and conflicts. However, even if every anomaly represents a potential impact to digital experience, it might not have a relevant effect on employees. The anomalies of interest are in fact those that affect a large number of devices, thus impacting a wide number of users, and those that persist over time, consequently having a continuous impact on *IDX*. In this perspective, it is necessary to quantify the digital experience impact as soon as anomalies are identified, and to prioritize the ones that most negatively affect the *IDX* score. In this way, anomalies can be classified according to their *IDX* impact.

Since *IDX* is a third-party index with a fixed refresh rate, it is not possible to immediately know the *IDX* score from raw metrics recorded values in a critical situation, so the impact assessment task of an anomaly is a major challenge. In addition, a related concern consists in estimating *IDX* impact of anomalies in near future, thus allowing for proactive corrective actions, before regressions can have devastating impacts.

Root cause analysis. A third challenge consists in analyzing the causes of the identified anomalies threatening the employees digital experience. Software changes (*i.e.*, updates, installations and removals) are millions per month. This makes the identification of changes responsible for an anomaly an expensive task that requires wide analysis. Indeed, each of the 30,000+ devices has its own configuration involving thousands of installed applications, so that each one triggers specific software updates and different potential conflicts can come out. In addition, the instant when a software change is performed on a specific application may vary from device to device. In that, the monitoring team deals with a huge amount of data to analyze. Currently, internal root cause analysis practices rely on expensive and manual human-based heuristics and statistical analysis.

6.3 Proposed solution

Figure 6.2 presents an overview of *RADig-X*, a tool for regression analysis of employees digital experience, which includes three major components, each supporting a targeted challenge: (i) *Anomaly Detection*, (ii) *Anomaly Impact Estimation* and (iii) *Root Cause Analysis*.

The first component of *RADig-X* leverages machine learning models aiming to detect anomalies. Thereafter, linear regressors are used in the second component for estimating the digital experience impact of each identified anomaly of runtime metrics. The third component supports Root Cause Analysis (RCA) activity by analyzing, through heuristic ranking, the impact of software updates on the digital experience. Across the three components, the tool employs an underlying model maintenance layer that is responsible for updating employed models with incoming data.

The whole tool lays on a external *Data Ingestion* layer that contains all employees devices monitored data, which is continuously populated by a monitoring agent.

RADig-X offers a clear separation of concerns in terms of challenges addressed by individual components separately. Indeed, in the daily practice, *RADig-X* components can be used either sequentially, to automate the whole process, or individually to support a single challenge (*e.g.*, root cause analysis).

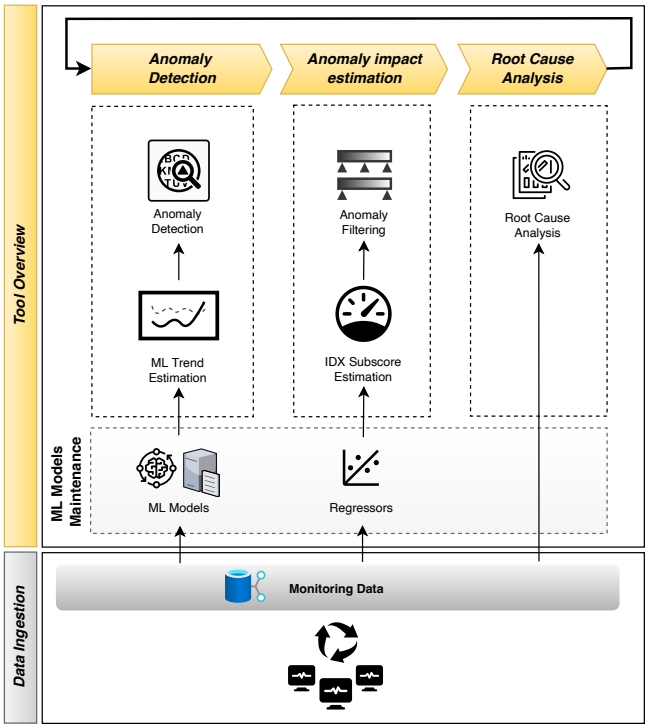


FIGURE 6.2: Overview of *RADig-X*

In the following we describe, in detail, how our solution addresses each of the detected challenges.

6.3.1 Anomaly detection

We employ time series ML forecasting algorithms (*e.g.,* LSTM) to model runtime metric fluctuations patterns based on the historical trends, as represented in Figure 6.2 (*ML Trend Estimation*). Specifically, one model has been trained for each time series of monitored runtime metric. In this way, we allow *RADig-X* to learn from the underlying seasonal patterns of each monitored runtime metric, thereby enabling the prediction of expected future values within a predefined forecasting horizon. In order to perform anomaly detection, predictions for expected runtime metric values are continuously generated by *RADig-X*, and the residuals between the expected and actual values are assessed to quantify the deviation from the expected time series behavior.

These deviations can be assessed in two different ways: (i) using a predefined threshold T, which enables automated anomaly detection, and (ii) using a dedicated dashboard that allows the monitoring team to visually compare the actual runtime metric trend with the expected one. Figure 6.3 shows an example of such visual analysis. On the left side, a solid line shows the real (standardized) number of crashes observed in two weeks, while a dashed line represents the predictions. The plot on the right side of the figure instead reports the deviation between the actual and predicted values. This example shows that in *day 10* there is a potentially impactful detour, and the monitoring team can decide whether to deem it as an anomaly based on their domain expertise.

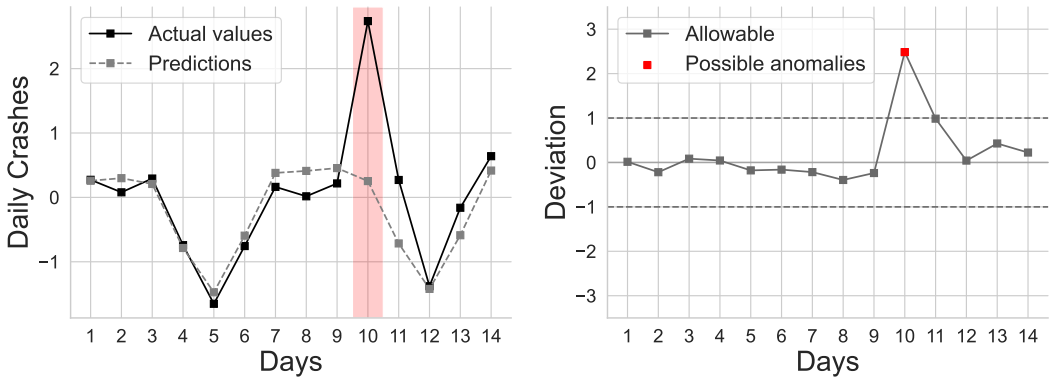


FIGURE 6.3: Real example of plots included in *RADig-X* visual anomaly detection dashboard.

6.3.2 Anomaly Impact Estimation

Our second challenge consists in pointing out anomalies with a relevant impact on digital experience. The capability to have a *IDX*-estimation strategy permits to assess the digital experience impact of runtime metrics anomalies without waiting for next *IDX* refresh, as motivated in Section 6.2. In order to point out relevant anomalies, *RADig-X* aims to estimate the *IDX* subscore of each considered metric starting from runtime metric values. In that, the *IDX Subscore Estimation* component of Figure 6.2 relies on linear regressors fit with the past observed data using raw metric values as input and their associated *IDX* subscores as target values. This technique allows to provide an estimation of *IDX*, while ignoring the details behind its calculation.

As reported in Section 6.1, global *IDX* score is based on a subset of runtime metrics, each one associated with a *IDX* subscore. Taking into account the fact that global *IDX* derives from several runtime metrics distinct in their characteristics, our solution estimates *IDX* subscore of each individual metric separately, and then performs the aggregation through a weighted average according to the *IDX* configuration established by the company. Indeed, each metric may have different units of measurement (*e.g.*, time units, percentage, etc.) and may be influenced by different factors (*e.g.*, device performance, user behavior, etc.). In that, *RADig-X* aims to estimate the impact of runtime metric anomalies on digital experience, by means of the global *IDX*. The *Anomaly Filtering* component, represented in Fig. 6.2, uses the impact estimation to filter out negligible anomalies (with a very low impact).

Furthermore, when used in tandem with the ML models presented in the previous section, this approach can also be used to predict future estimations of the global *IDX*, and estimate the future impact on digital experience of current runtime metric fluctuations.

6.3.3 Root Cause Analysis (RCA)

The huge amount of data collected from monitored devices makes the root cause analysis (currently conducted manually) a labour-intensive and time-consuming task. Our proposed solution implements a heuristic ranking, by automating a before-and-after analytical process on all devices. This step is represented in Figure 6.2 with the *Heuristic Ranking* component. This process is aimed at automatically narrowing down the many possible causes to a few changes, with respective quantitative and

qualitative impact measures. A major challenge consists of managing devices with heterogeneous software configurations, which execute updates in different instants determined by the user.

Figure 6.4 shows a high-level description of *RADig-X* Root Cause Analysis (RCA). The analysis starts from the anomaly report generated by the previous tasks containing: (i) the time interval in which an anomaly has been observed, (ii) the software application that is crashing, and (iii) all the details related to the devices that are affected. *RADig-X* extracts all the data related to software changes and software crashes collected by the agent installed on the devices, basing on the time interval in the anomaly report. In particular, software changes data are extracted as a dataset, in which each row corresponds to a software change event and specifies: device ID, software name, software version and timestamp. Similarly, software crashes data are collected as a tabular dataset, where each row identifies a software crash recorded by the agent, and it specifies: application name, device ID and crash timestamp. We also include a filtering step to possibly remove non-relevant data (*e.g.,* changes concerning very few devices).

Upon completion, the tool executes a loop running through all the unique software changes contained in the software changes dataset. Within each iteration (*i.e.,* each unique software change), a nested loop is performed over all the devices that received that specific change. For each device, *RADig-X* splits the crashes dataset of that device in two parts, namely: before SC subset and after SC subset. Statistical metrics are then carried out, for each software change, by comparing before/after events. Finally, generated data are sorted out on a parameter that summarizes the global impact of the software changes on all the devices.

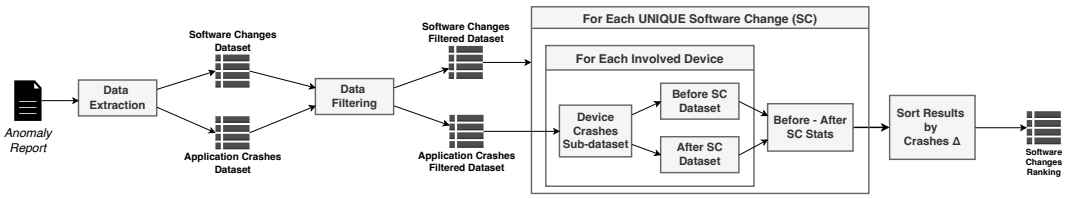


FIGURE 6.4: *RADig-X* Root Cause Analysis (RCA) process diagram

6.4 Empirical Evaluation

The overall evaluation is conducted through independent assessments of the effectiveness of each *RADig-X* sub-component (*i.e., anomaly detection*, *anomaly impact estimation* and *RCA*), by aiming to address the following research questions:

- **RQ**$_{5.1}$: How effective is *RADig-X* in detecting anomalies?
- **RQ**$_{5.2}$: How effective is *RADig-X* in assessing the relevance of anomalies in terms of employees digital experience?
- **RQ**$_{5.3}$: How effective is *RADig-X* in supporting root cause analysis?

The results of our empirical evaluation are reported in the following subsections, each one dedicated to a specific research question.

6.4.1 $RQ_{5.1}$: How effective is *RADig-X* in detecting anomalies?

To address this research question, we first assess the feasibility of different machine learning models in predicting time series of runtime metrics. Then, we evaluate the effectiveness of these ML models for anomaly detection, through a comparison with the current internal practice.

To select *RADig-X* ML model for performing anomaly detection, we tested the effectiveness of two widely-used Recurrent Neural Networks (RNN) models, namely *bidirectional Long Short-Term Memory (LSTM)* [221], [299] and *Gated Recurrent Unit (GRU)* [219], employing *Random Forest Regressor* (RFR) [300] and *Linear Regression* (LR) as baseline models for comparison, since they have been widely employed for time series forecasting [204], [301], [302], [303], [304]. To construct the evaluation dataset the monitoring team has suggested eight metrics concerning application crashes with higher weights in *IDX* configuration, and so expectedly most impactful on *IDX* in case of regressions. We extracted more than one year of data with a sampling interval of one day for each metric, generating eight time series with *400* values. We then formulated time series forecasting task as a supervised learning regression problem. The dataset has been crafted creating consecutive shifted windows, each with a width of 28 time steps, for each time series. For each window, the first 14 time steps are used as input and the last 14 values as output. We adopted a multiple-output strategy for addressing the multi-step-ahead problem, thus directly predicting all the forecasting horizon [299], [305]. For each windowed dataset that has been generated, first 80% of windows are used for training (of which 10% for validation set) and the last 20% as testing set.

In order to identify the most effective ML approach for our dataset, we leveraged two widely used forecasting accuracy measures: (i) Mean Absolute Error (*MAE*) and (ii) Root Mean Squared Error (*RMSE*) [204], [306], [307] over the testing set. We have implemented *LSTM* and *GRU* using *Keras*. Linear Regression and Random Forest Regressor models have been implemented by using *scikit-learn*.

The effectiveness of *RADig-X* anomaly detection has been evaluated over the testing set of the same time series. For sake of such comparison, we have implemented an automated script to replicate the current anomaly detection practice used within the company. Specifically, current internal practice estimates future metric values computing the mean value based on the specific weekday past values. For instance, expected metric value for the next Monday is computed averaging values recorded on previous Mondays, considering a time window defined by taking into account retention policies on monitored data. Daily anomaly detection is then performed by comparing the expected values and the real observed ones with threshold-based criteria.

We computed *Precision*, *Recall* and *F1-score* using as ground-truth actual anomalies occurred on the selected time series, as reported by the monitoring team.

RADig-X and current internal practice anomaly detection are both performed by comparing the difference among expected and observed values against a threshold. To fairly compare them we used the same threshold, setting it equal to the standard deviation σ of the input sub-sequence: $T = \sigma$. We chose this conservative threshold based on the feedbacks of the monitoring team. Indeed, the monitoring team tends to consider false negatives more dangerous than false positives. In this context, an anomaly that goes unnoticed can be more damaging than a false positive, which instead might be double checked by the team and ignored afterwards.

Results. Figures 6.5 and 6.6 depict the RMSE and MAE distributions for each ML approach. Given the presence of outliers in error distributions and the variety of

applications considered, we preferred to use the median values to compare the approaches. We noticed that RNN models exhibited favorable MAE distributions, with lower mean and median with respect to LR and RFR. Specifically, in Fig. 6.6 we observed a median of 0.071 for LSTM and 0.072 for GRU, while LR and RFR have shown median values of 0.083 and 0.093, respectively. By looking at Fig. 6.5, we again noticed that all RNN models exhibited a lower RMSE median compared to the baselines. Namely, RMSE median of 0.131 and 0.121 have been observed for LSTM and GRU, versus 0.137 of LR and 0.139 of RFR. In summary, both MAE and RMSE medians pertaining to GRU model are the lowest ones. In addition, the GRU error distribution is less spread out, thus suggesting a more reliable prediction. Based on these quantitative results, we have chosen LSTM and GRU models for the implementation of *RADig-X*, and therefore for the subsequent evaluation step.

In the second step of $RQ_{5.1}$ evaluation, we compared *RADig-X* anomaly detection effectiveness against current internal practices. The evaluation has been performed over the same eight time series, by using anomalies manually labeled by the monitoring team as ground-truth. Results of anomaly detection evaluation are shown in Figure 6.7, where we present the distribution of *Recall*, *Precision* and *F1-score*, as grouped by approach. Specifically, we compared the following approaches: (i) current internal practice, (ii) LSTM-based *RADig-X* and (iii) GRU-based *RADig-X*. By examining the F1-score boxplots, we observed highest mean and median when using LSTM model, with a mean of 0.437 and a median of 0.408. Instead, GRU exhibited a median of 0.39 and mean of 0.30. With the current internal practice, we noticed a median F1-score value of 0.225 and a mean of 0.331. Thus, the median values show an advantage in using LSTM. Interestingly, even though the GRU model has shown a lower error during the previous model performance evaluation, the quality of *RADig-X* anomaly detection is better when using LSTM, which highlights a *median* F1-score improvement of **94%** compared to the current internal approach. By examining *precision* and *recall* separately, LSTM-based *RADig-X* has shown an improvement of **40%** in median *recall* compared with the current internal approach, suggesting that *RADig-X* detected considerably more true anomalies. In addition, a larger improvement in median *precision* (**100%** increasing) is also achieved, meaning that LSTM-based *RADig-X* significantly improved the number of actual anomalies among those identified. By considering the improvements presented above, we clearly observed that the proposed strategy improves the accuracy of the anomaly detection technique.

6.4.2 $RQ_{5.2}$: How effective is *RADig-X* in assessing the relevance of anomalies in terms of employees digital experience?

Second RQ aims to address the effectiveness of *IDX* impact estimation of the identified anomalies.

Our strategy leverages linear regressors with a supervised learning approach, by using runtime metrics data as input and the corresponding *IDX* subscore as output, in order to estimate a global *IDX* value. We have considered all the runtime metrics accounted for *IDX* calculation (that are more than 60). Specifically, we built a dedicated linear regression model for each runtime metric, and performed a separate training for each one. We implemented linear regressors using *scikit-learn* in *python*. Each regressor has been trained using time series of runtime metric weekly values as input, and the corresponding *IDX* subscores time series as output. Linear regressors aim to predict *IDX* subscore of a specific runtime metric, starting from the metric weekly value. For each metric, the regressor has been fit using the first 80%

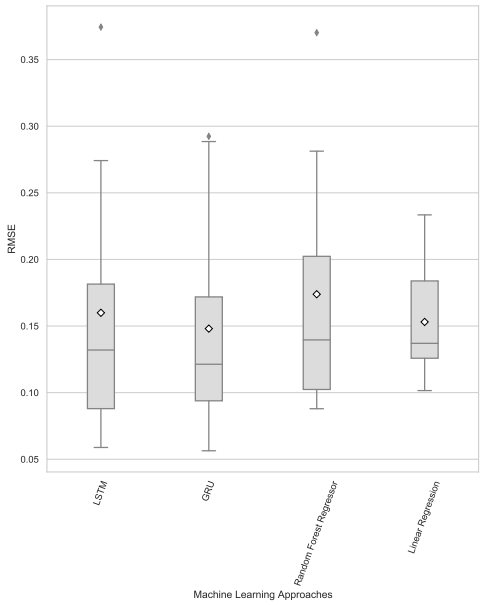


FIGURE 6.5: RMSE of ML Models.

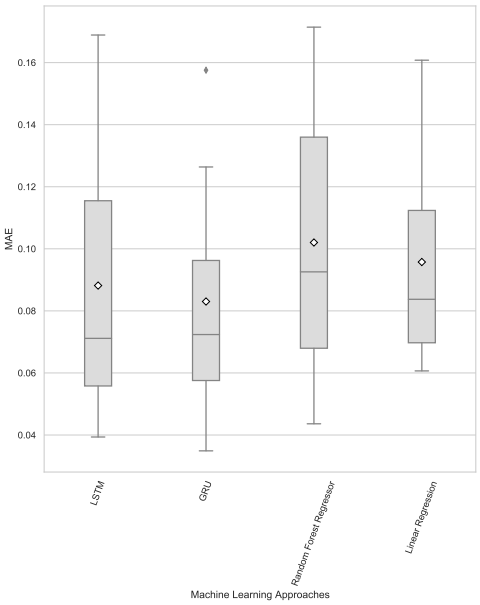


FIGURE 6.6: MAE of ML Models.

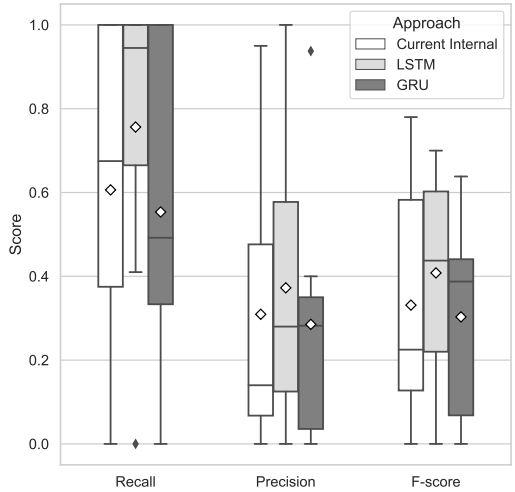


FIGURE 6.7: Comparison of *recall*, *precision* and *F1-score* scores distribution for all anomaly detection approaches.

of time series as input and the last 20% for testing. We used the *Root Mean Squared Error (RMSE)*, rescaled on a 0-100 scale using the minimum and maximum values of *IDX*, to evaluate the estimation quality of each runtime metric. In order to test the effectiveness of the global estimation, we have then aggregated all the metric *IDX* subscores to obtain the global *IDX* estimation. This outcome has been compared against the real global *IDX* score observed in the testing set. Predicted *IDX* subscores are aggregated according to the weights that the company assigned to each runtime metric in *IDX* configuration. Based on company requirements, a global *IDX* estimation percentage error less than 5% is acceptable.

Summing up, $RQ_{5.2}$ evaluation has been conducted in two steps: (i) measuring linear regressors error to assess that the error distribution is acceptable across all the metrics, and (ii) evaluating the effectiveness of the aggregation technique of estimated scores by comparing the aggregated predicted score with the real global *IDX* score.

Results. Figure 6.8 shows the distribution of all the regressors absolute errors. We highlight that Q1 is very close to *0* and Q3 is less than *5*, *i.e.*, 75% of regressors have an error lower than *5*. Hence, the majority of regressors have a very small error. However, the plot also shows several outliers, thus suggesting that some components are too complicated to be modeled with a simple linear regressor.

Table 6.1 shows the global *IDX* estimation percentage error in the last *7* weeks. Global estimation error is always less than *2%*, thus meaning a suitable estimation technique based on company requirements.

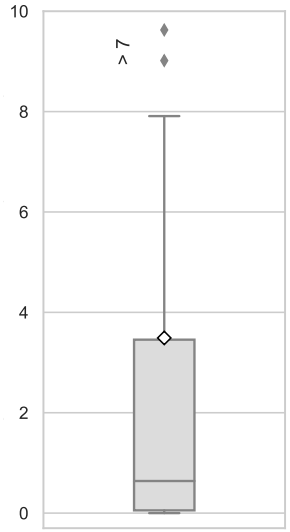


FIGURE 6.8: *RMSE* distribution in *IDX* subscore estimation over all considered runtime metrics using linear regressors.

6.4.3 $RQ_{5.3}$: How effective is *RADig-X* in supporting root cause analysis?

The RCA strategy has been empirically evaluated by applying it in real-world scenarios, specifically in four cases where significant application crash anomalies occurred. The data we used for the evaluation consist, for each scenario, in application crashes and software change events, occurred in each device, which have been collected by APM tool agent. For each analysis, we have generated a ranking of software changes in a decreasing order of impact. Software modules and application names have been anonymized for privacy reasons. The naming convention is the

Testing Week	*IDX* Percentage Error
Week #1	0.33 %
Week #2	0.08 %
Week #3	0.54 %
Week #4	1.36 %
Week #5	0.39 %
Week #6	0.12 %
Week #7	0.79 %

TABLE 6.1: Percentage error measured on weekly global *IDX* estimation.

following: software updates are indicated with the prefix *SU_*, while applications that have experienced a crash regression are denoted by the prefix *CR_*. Software update modules that pertain to the same application share the first letter after the prefix (*e.g.*, SU_A.1 and SU_A.2 in Tab. 6.2 are both related to the application A).

As for ranking metrics, we considered for each software change: (i) *Crashes Difference*, representing the relative difference between the total amount of crashes observed before and after the software change occurred on all the affected devices; (ii) *Devices*, which indicates the percentage of devices affected by the change; (iii) *Crashes Delta Avg*, calculated by averaging the difference between the crashes observed before and after the change of each affected device. The sorting of software changes ranking is performed in descending order based on *Crashes Delta Avg* value.

Results. In the first scenario, an application that we name as *CR_X* has experienced an increase of +450% in the number of crashes. In Table 6.2, we reported the head of the ranking generated by *RADig-X* RCA. The top-ranked updates clearly depict major crashes difference increases (by more than +1,000%), all related to the same application *SU_A*.

Other two scenarios concern a common application that we name as *CR_Y*. One of them has manifested a notable crash regression in *CR_Y*, whereas in the other one there was a subsequent decrease in the number of crashes for the same application. The results are shown in Tables 6.3 (crash increasing phase) and 6.4 (crash decreasing phase), respectively. By looking at *Crashes Difference* and *Crashes Delta Avg* column of Table 6.3, we can easily infer that *SU_G.1* and *SU_G.2* updates result in a much stronger impact than other ones. Similarly, Table 6.4 shows that the first row has a more significant crash delta average value (-22.18%) than the following ones. By looking at the software updates names, we find that these updates actually concern different modules of the same application update (*SU_G*), thus suggesting that *SU_G.1* and *SU_G.2* updates of Tab. 6.3 have caused regressions that might have been fixed by *SU_G.6* update in Tab. 6.4.

In the fourth scenario, which involved another crash regression, the heuristic ranking did not report any meaningful results.

In summary, in three out of the four scenarios the tool has highlighted a subset of software changes that may have caused the crashes regression. However, after consulting updates support team, we discovered that only in the first scenario (Table 6.2) the software change that had really caused the regression has been highlighted. In contrast, the regressions in the other cases were attributed to particular workload distributions in the industrial setting and not to specific software changes. Hence, the approach has proved its usefulness only in the first case. One critical issue that

has been found is the distortion of results on some software changes that occur simultaneously with the actually responsible one. Since the approach counts events before and after updates, if a group of changes is made at the same time then the consequences of one change belonging to the group could bias the results of the whole group. On the basis of the above consideration, we can conclude that the approach succeeds in readily filtering out the subset of software changes that impacted the devices most significantly, but the identified changes may not be the real culprits since crashes regression might be mainly due to some other reasons.

Software Update	Crashes Difference	Devices	Crashes Delta Avg
SU_A.1	+ 1205.27 %	58.59 %	5.49 %
SU_A.2	+ 1205.27 %	58.59 %	5.49 %
SU_A.3	+ 1209.29 %	58.62 %	5.49 %
SU_A.4	+ 1205.27 %	58.59 %	5.49 %
SU_A.5	+ 1206.36 %	58.59 %	5.49 %
SU_A.6	+ 1201.99 %	58.57 %	5.48 %
SU_A.7	+ 1206.78 %	58.43 %	5.46 %
SU_D.1	+ 345.99 %	80.20 %	4.03 %
SU_E.1	+ 266.42 %	100.00 %	3.45 %
SU_E.2	+ 233.08 %	89.00 %	3.37 %
SU_E.3	+ 146.55 %	61.28 %	3.02 %
SU_F.1	+ 30.28 %	95.79 %	1.01 %
...			

TABLE 6.2: *RADig-X* root cause analysis evaluation: ranking for CR_X crash increase.

Software Update	Crashes Difference	Devices	Crashes Delta Avg
SU_G.1	+ 189.60 %	74.87 %	52.10 %
SU_G.2	+ 199.23 %	98.04 %	39.11 %
SU_G.3	+ 40.81 %	69.41 %	8.34 %
SU_G.4	+ 27.23 %	72.19 %	4.97 %
SU_G.5	+ 27.11 %	72.61 %	4.96 %
...			

TABLE 6.3: *RADig-X* root cause analysis evaluation: software updates ranking scenario 2 (CR_Y crash increase).

Software Update	Crashes Difference	Devices	Crashes Delta Avg
SU_G.6	- 51.24 %	92.43 %	- 22.18 %
SU_G.7	- 20.92 %	76.80 %	- 5.78 %
SU_G.8	- 17.28 %	95.71 %	- 4.63 %
...			

TABLE 6.4: *RADig-X* root cause analysis evaluation: software updates ranking scenario 3 (CR_Y crash decrease).

6.5 Insights from practice

RADig-X has been tested and deployed in Micron, used daily by monitoring team analysts. We report two success cases collected thus far from the adoption of the tool within the company, focusing on how *RADig-X* assisted the monitoring team in managing runtime metrics anomalies that have affected *IDX*.

In either cases, a first anomalous increasing phase in application daily crashes occurred followed by a later decreasing phase. Thanks to the visual dashboard provided by *RADig-X* (that includes plots like the ones shown in Figure 6.3), the monitoring team can double check and analyze the anomaly, observing more details about the deviation between expected and observed values to perform in-depth analyses. Both cases start with an anomaly reported by *RADig-X*, which is immediately analyzed by the monitoring team via the visual dashboard. Right away, by leveraging *RADig-X*, the real-time impact of the anomaly on *IDX* is estimated (without waiting for the next *IDX* refresh), thus allowing the monitoring team to immediately know the impact of the anomaly on the digital experience. Then, the extraction of crashes and software changes data collected during the anomaly-affected week is performed to generate the heuristic ranking with *RADig-X*, thus filtering the most impactful software changes.

We summarized quantitative data of two cases in Tables 6.5 and 6.6, showing the observed deviation calculated as the absolute percentage error between the collected actual values and the predicted ones in the metric time series. The error is computed on both the sum and the mean of the metric daily values. In addition, we report the anomaly percentage impact on the *IDX* metric subscore that incurred the anomaly.

Case 1: Results concerning the first case are presented in table 6.5. *RADig-X* anomaly detection dashboard showed a difference of more than 60% between the expected and observed crashes of a running application *APP1*[5]. Such immediate recognition of the anomaly allowed for preliminary investigations to figure out the underlying concern. By comparing the total amount of *APP1* crash events occurred in all the devices in the week before the anomaly with respect to the week including the anomaly, the monitoring team noticed a 400% increase. By more closely looking at the devices that are plagued by the anomaly, it turned out that the problem was distributed on almost all the devices of the company, thus impacting the digital experience of many employees. Indeed, *RADig-X* estimated an impact to *APP1 IDX* subscore of 25.23%. Thereafter, *RADig-X* RCA ranking highlighted an impactful software change *SC1*. By summing all the application crashes observed in each individual device after this software change, it has been found that the number of *APP1* crashes after *SC1* was 16 times the number of crashes before the change (considering a one week length window before and after the change). The *APP1* anomaly and the likely cause *SC1* have been immediately reported to support groups to fix the issue. After three months, the *RADig-X* signaled another *APP1* anomaly concerning a decreasing trend in the amount of daily crashes of the same application. The gap between predicted and observed was about 35%. Even though this situation was supposed to increase *IDX*, by exploiting *RADig-X* the team noticed that the increase was not enough to raise the *APP1 IDX* subscore. *RADig-X* RCA ranking has shown a potentially critical software change *SC2* (which was a module of *APP1* itself). By comparing the sum of daily crashes occurred in all the devices before and after *SC2* (considering a one week period), the number of *APP1* crashes decreased by 30.2%.

	Observed deviation (%)		
	Sum	Mean	*IDX* Subscore
Regression Phase	64.51 %	64.50 %	- 25.23 %
Improvement Phase	35.99 %	36.03 %	0 %

TABLE 6.5: *RADig-X* practical usage case 1 results.

[5]For privacy reasons, we cannot disclose the name of the actual applications and software changes.

Case 2: The second situation is reported in Table 6.6. By using the tool, the monitoring team identified an anomaly in *APP2* daily application crashes owing to an unusual growth pattern for observed crashes. The average gap between daily expected and observed values was over 200%. Then, *RADig-X* estimated a *IDX* subscore impact of almost 50%, thus allowing the monitoring team to be aware in real-time of the impact of the anomaly on the employees digital experience. *RADig-X* RCA, in this case, didn't highlight any specifically related impacting software update. After two months, *RADig-X* detected a decreasing in daily *APP2* crashes, thus signaling the anomaly to the monitoring team. On average, the percentage error among expected and observed daily crashes decreased by 11.11%. By using *RADig-X*, the monitoring team immediately estimated an improvement in *APP2 IDX* subscore of 69.37%.

In these two situations we observed the real utility of the tool in a practical context. Indeed, the monitoring team confirmed that these two situations consistently impacted the employees digital experience. Thanks to *RADig-X*, it was possible to: (i) immediately identify runtime metrics anomalies, (ii) estimate in real-time the digital experience impact without waiting for *IDX* refresh, and (iii) identify updates that exhibit a stronger correlation with the metric anomaly. In addition, we observed the effectiveness of *RADig-X* in detecting anomalies not only in the case of *IDX* regressions but also in the case of improvements. As a matter of fact, albeit they do not have a negative impact, they are still of interest to the team.

The context of the tool concerns third-party applications analysis (in both anomaly detection and software changes analysis), therefore it is not possible to investigate in detail which types of software modifications (refactoring, integration or others) led to crashes and which code fragments have been affected.

	Observed deviation (%)		
	Sum	Mean	*IDX* Subscore
Regression Phase	236.25 %	236.34 %	- 48.9 %
Improvement Phase	58.46 %	11.11 %	+ 69.37 %

TABLE 6.6: *RADig-X* practical usage case 2 results.

We also measured the global *IDX* since the adoption of *RADig-X*, resulting in a 14.66% improvement. In addition, the monitoring team remarked that the use of the system automatically implies more focus on the monitored data, leading to a greater sensitivity to any variation.

6.6 Threats to Validity

Construct validity We evaluate the effectiveness of each component of *RADig-X* independently from the other ones. The interactions among the different *RADig-X* components might affect the approach effectiveness. Unfortunately, due to APM tool retention policies, we were unable to conduct a comprehensive end-to-end evaluation of the approach. Nonetheless, in Section 6.5 we report some of the benefits of employing *RADig-X* in practical scenarios. We have replicated the current practice used within the company through an automated script. Although this script may not precisely represent the monitoring team behavior in practice, it serves as a valuable baseline for evaluating *RADig-X*.

The anomaly detection component of *RADig-X* has been evaluated on eight distinct time series of runtime metrics. The effectiveness of *RADig-X* may vary when applied to other runtime metrics. Unfortunately, the selection of the time series used

in this evaluation has been constrained by the retention policies of APM tool and the necessity of a ground-truth, *i.e.,* anomalies manually labeled by domain experts.

Internal validity We evaluate *RADig-X* using well-established metrics, such as MAE and RMSE. Using different evaluation metrics could impact the interpretations of the results. *RADig-X* determines the presence of an anomaly based on a specific threshold T. The outcomes of *RADig-X* may vary if different threshold values are applied. To ensure a fair comparison, we have based our comparison between *RADig-X* and the baseline on the same threshold.

External validity We use *RADig-X* within the context of Micron in combination with a specific APM tool. However, we designed *RADig-X* with a focus on generalizability, as we aim to facilitate its application to other contexts. Indeed, if the semantics of the data is the same as the one used in the Micron context, then the effort to adapt this approach would be related only to the data formatting. Of course, its effectiveness may be affected by the context in which it is used.

6.7 Related work

Software Anomaly Detection. In recent years, researchers have increasingly utilized machine learning techniques alongside traditional statistical approaches (*e.g.,* ARIMA models [308], [309]) for software anomaly detection [52], [55], [59], [60], [61], [62]. Among the most commonly employed models are Recurrent Neural Networks [55], [310], [311], [312], such as LSTM and GRU. For instance, LSTM have been employed to predict failures in cloud service systems [313], or spacecraft anomalies [52]. LSTM have been also combined with other techniques such as Variational Auto-Encoder (VAE) [314] and multimodal learning [315] to perform this task. Other methods utilize other unsupervised machine learning approaches to detect anomalies [47], [316], [317]. Chen et al. [318] use pattern sketching to detect anomalies in online service system, a technique that provides better result interpretability when compared black box approaches. Most of the prior work apply anomaly detection to individual (in-house) software applications. Our work, instead, involves dealing with a large IT infrastructure that encompasses a variety of third-party software applications spanned across more 30,000 devices.

Software Changes. Software quality degradation may be induced by software changes (such as bug fixes, refactoring, functionality extension). For this reason, over the last decade, researchers have been committed in devising techniques to mitigate the impact of software changes. These techniques can be preventive, *i.e.,* before the changes are deployed into production to evaluate potential risks [319], [320], [321], [322], or post-deployment [55], [119], [120], [121]. Our approach fits into this last category. Existing approaches of this category are based on examining variations on quality metrics before and after deployment. For instance, Lumos [323] adopts an A/B testing approach (similarly to our heuristic) while FUNNEL [121] leverages singular spectrum transform (SST) algorithm. Other approaches such as Gandalf [120] and SCWarn [55] have been realized to take into account data monitored from multiple sources, such logs and a variety of KPIs. Much of these approaches are focused on analyzing software changes related to a specific service or application. Our study, instead, proposes an analysis on software changes related to third party applications, considering the software change as a black box event.

Root Cause Analysis. Root cause analysis (RCA) of software systems has been extensively researched in a variety of contexts. For instance, DeLag [57] automatically detects deviations in the execution time of Remote Procedure Calls (RPC) that are correlated with performance degradation, in the context of service-based systems. In the same domain, automated pattern detection has been widely investigated for root cause analysis of performance issues and/or failures [324], [325], [326]. In StackMine [327], pattern detection has been employed to detect callstack patterns associated with performance issues. Chen et al. [145] proposed a failure cause diagnosis approach using decision trees trained over request traces in which there are failures. *RADig-X* seeks to identify software changes that are associated to significant digital experience regressions.

6.8 Conclusion

In this work, we have introduced *RADig-X*, a novel approach to enhance the diagnosis and root cause analysis of digital experience issues. We have conducted an empirical evaluation on real-world monitoring data gathered from a large company, involving over 30,000 devices. The results have shown that *RADig-X* can be effectively used to (i) detect anomalies in runtime metrics, (ii) assess the relevance of anomalies with respect to their impact on the global *IDX*, and (iii) identify software updates that are likely culprits of anomalies. Since its deployment, *RADig-X* has helped to identify problematic software updates that have shown relevant impact on the digital experience of Micron employees. As a future work, we intend to broaden the assessment of *RADig-X* to incorporate other types of runtime metrics, such as those related to software performance. Furthermore, we plan to evolve *RADig-X* beyond its current capabilities. In particular, we plan to shift towards a multi-feature approach in the anomaly detection task, and to include the capability of proactively identifying potential problems with software updates before they are widely deployed across numerous digital devices.

Chapter 7

Conclusions

In this thesis, we presented a set of contributions that empirically demonstrate the effectiveness of AI-driven approaches for supporting software performance engineering (SPE), providing valuable insights into their strengths and limitations in this context. Each chapter of this thesis has been enriched with a threats to validity subsection highlighting conceptual and technical limitations of each presented approach. In the following, we recall some limitations with the goal of explaining how they open up promising future research in this area.

7.1 Limitations and Future Work

7.1.1 AI for SPE at Development Stage

State-of-the-art language models. As of now, an ever-growing plethora of language models is continuously developed and released. Many of them have different architectures and are trained in different ways. Hence, different language models may yield different results [84], especially when models are scaled up [328]. Specifically, state-of-the-art language models demonstrate superior performance in optimizing source code [329], [330]. On this side, a possible future direction is to broaden our analysis by using a more diverse set of models in architecture and scale, including state-of-the-art models.

Real-world code optimization. We evaluated our execution-aware strategy over source code from CodeNet [152], consisting of self-contained programs to solve specific programming challenges. Instead, when dealing with real-world complex systems involving modules and sub-modules, libraries and intricate interdependency, some additional challenges emerge, like trying to optimize methods or classes beyond the provided contexts [79]. Recent studies started indeed to move from self-contained source code towards real-world repositories optimization [330], [331]. It would be interesting to experiment the effectiveness of introducing code execution aspects in such complex contexts.

7.1.2 AI for SPE at Testing Stage

Generalization across diverse JIT-equipped VMs and hardware. An initial warmup phase is a defining characteristic of all JIT-equipped virtual machines, as they have to initially spot frequently executed ('hot') code before the subsequent just-in-time compilation. However, different VMs might perform this task differently [332], [333], thus resulting in different time-series patterns. Furthermore, since the compilation translates into machine code, the hardware on which the tests are executed plays a key role in this phase. Exploring AI-driven steady-state detection techniques

by considering different VMs—including those designed for other programming languages—and varying the execution hardware might reveal interesting insights into the portability of this approach across similar yet different contexts.

Sensitivity analysis to testing parameters. The parameters used to set the microbenchmarking task, such as the number of forks and the measurements iterations, can influence the quality of results and the testing time of microbenchmarks. For instance, a too low number of measurements iterations may generate a time series which is not long enough to have a comprehensive visibility of the warmup end. Employing an AI-driven strategy involves additional parameters to set in the testing infrastructure. For instance, the (sliding) window length must be chosen to ensure that exploitable information (warmup-specific patterns in our case) is captured, as shown in other time series analysis work [334], [335], [336]. Carrying out a further sensitivity analysis on such parameters would shed light on their impact on the classification effectiveness.

7.1.3 AI for SPE during Post-Deployment

Extension of the TSF-related findings to different software metrics. The results related to the effectiveness of TSF methods in forecasting software runtime metrics may not generalize to other different software runtime metrics. We aimed to mitigate this threat by including hundreds of different software runtime metrics collected from 29 distinct software applications, which span seven distinct metric classes. However, most of the metrics that we have analyzed are related to software latency and application crashes, while other resource-related metrics are also pivotal in continuous software monitoring, such as memory consumption and CPU utilization. This represents a promising future direction to verify whether the effectiveness of modern TSF methods extends to such metrics and to scout possible characterizations related to them. Additionally, we only focused on daily software runtime metrics, while other scenarios may require dealing with metrics that have different levels of aggregation. Another interesting direction is to investigate whether our results generalize to other types of time aggregation granularity.

Context-specific data. We studied the effectiveness of TSF approaches for forecasting runtime software metrics belonging to a wide real-world infrastructure. Despite the vast amount of devices involved in this domain, the findings may not generalize to other contexts having different device monitoring policies and usage profiles. We encourage future investigation of TSF approaches across diverse contexts to better assess their effectiveness in software monitoring.

7.1.4 On the hidden costs behind AI-driven SPE

Modern AI-enhanced approaches represent a promising solution for supporting and improving the current software performance engineering practice. However, adopting AI-driven techniques involves significant costs.

Firstly, the latency induced by the inference stage of modern deep-learning models could pose a risk when using them in performance-critical contexts. This could be the case of adopting AI-driven anomaly detection techniques in a real-time monitoring scenario, where the detection strategy might have a very strict requirement in terms of response time. Another example is the introduction of AI-enabled classifiers in the testing stage, where using AI-driven techniques for dynamically halting the testing process requires to generate a single prediction for each new measurement

iteration. Depending on both the model in use and the surrounding testing environment, this practice could greatly influence how long testing takes. This sets the stage for a thorough exploration to uncover the leanest model that can still deliver accuracy worth trusting. However, this task requires additional effort and domain-expertise.

Secondly, deep-learning models require to be properly trained. The training stage often is computationally expensive and, depending on the specific model, requires an adequate infrastructure. Even if training is typically done upfront, real-world deployments may eventually need retraining later to ensure the model adapts to data drift [337], [338]. Another key aspect is having access to enough data to build a reliable model with respect to the task to perform. One illustrative example concerns the labeled data for the anomaly detection task, which typically requires significant human effort and is characterized by an imbalanced distribution favoring normal (non-anomalous) data [312], [339]. Also, in complex production scenarios, retention policies needed to balance historical data and future storage may represent a risk to preserve the long-term history of the system behavior needed for properly training the model. At the same time, modern APM tools collect huge amounts of data, within which distinguishing training-relevant features from noise can be challenging [340], [341], [342].

7.1.5 Investigate AI for SPE beyond the software lifecycle phases

Software performance engineering activities also involve tasks that are not included in the three dimensions afforded in this thesis (*i.e.,* development, testing and post-deployment). Indeed, another dimension that has been widely explored in the performance literature are *model-based* approaches, which exploits performance models created in the early development stage to properly tune the architecture and design with the purpose of meeting performance requirements [18], [343], [344]. Modern language models can be investigated for supporting model-based architectural refactoring [345] focusing on performance requirements.

7.2 Epilogue

We provided a set of research contributions aiming to investigate the effectiveness of AI-driven approaches in supporting SPE, from early development to post-deployment, particularly focusing on real-world scenarios that introduce some additional challenges on their applicability. We empirically analyzed several existing and newly proposed AI-driven approaches in software performance context, helping researchers and practitioners to understand whether and how to introduce them in SPE. In the context of the development stage, we explored the impact of learning code execution aspects when automating source code optimization with language models. During the testing stage, we assessed the effectiveness of AI-driven methods in enhancing the current Java microbenchmarking practice. After deployment, we investigated the effectiveness of (statistical and AI-based) time series forecasting approaches for sake of proactive run-time software monitoring. We also explored the usage of meta-learning techniques for time series-based tasks in software engineering, most of them tightly linked to SPE practice, such as proactive software monitoring (TSF/TSAD) and steady-state performance detection (TSC). Finally, we reported the industrial experience carried out during the doctoral program. This falls into the post-deployment stage where digital devices are closely monitored for ensuring

smooth operability of employees. In that, we designed, implemented, and deployed *RADig-X*, an AI-enhanced tool for supporting digital experience regressions analysis of software/hardware systems in real-world scenario.

We critically examined the comparative effectiveness of each approach, whether purpose-designed or pre-existing, in this context. Overall, AI-driven approaches have emerged as a highly effective means for enhancing the current SPE practice, spanning stages from initial development to post-deployment. We discovered that each predictive approach is particularly effective on specific type of data, underlining the importance of carefully conducting an in-depth domain analysis and pushing towards the development of automated techniques for the best predictive algorithm selection based on the task and the context.

In conclusion, our findings revealed that AI-driven approaches yield promising results, successfully supporting software performance engineering (SPE) across development, testing and post-deployment phases. Such outcomes represent an encouraging starting point for many future directions in AI for SPE, a rapidly evolving research field with substantial unexplored potential.

Bibliography

[1] F. D. Menna, L. Traini, G. Bavota, and V. Cortellessa, "Investigating execution-aware language models for code optimization," in *33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, IEEE, 2025, pp. 204–215. DOI: 10.1109/ICPC66645.2025.00029. [Online]. Available: https://doi.org/10.1109/ICPC66645.2025.00029.

[2] 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, event-place: Sacramento, CA, USA, New York, NY, USA: Association for Computing Machinery, 2024, pp. 443–454, ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3695017. [Online]. Available: https://doi.org/10.1145/3691620.3695017.

[3] A. Trovato, L. Traini, F. Di Menna, and D. Di Nucci, "AMBER: AI-Enabled Java Microbenchmark Harness," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 762–766. DOI: 10.1109/ICST62969.2025.10988925.

[4] F. D. Menna, L. Traini, and V. Cortellessa, "Time Series Forecasting of Runtime Software Metrics: An Empirical Study," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024*, 2024, pp. 48–59. DOI: 10.1145/3629526.3645049. [Online]. Available: https://doi.org/10.1145/3629526.3645049.

[5] F. D. Menna, V. Cortellessa, M. Lucianelli, L. Sardo, and L. Traini, "RADig-X: A Tool for Regressions Analysis of User Digital Experience," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024*, 2024, pp. 359–370. DOI: 10.1109/SANER60148.2024.00043. [Online]. Available: https://doi.org/10.1109/SANER60148.2024.00043.

[6] A. B. Bondi, *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*, 1st. Addison-Wesley Professional, 2014, ISBN: 0321833821.

[7] E. Ryan, *I'll share 23 website load time stats to know: Conversions, mobile, and more*, 2023. [Online]. Available: https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/.

[8] H. Krasner, "The cost of poor software quality in the u.s.: A 2020 report," Consortium for Information & Software Quality (CISQ), Tech. Rep. CPSQ-2020, Jan. 2021. [Online]. Available: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf.

[9] K. Eaton, *How One Second Could Cost Amazon $1.6 Billion In Sales*, 2012. Accessed: Mar. 15, 2012. [Online]. Available: https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[10] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, 2000. DOI: 10.1109/32.888628.

[11] SITESPECT, *Case study: Mozilla optimizes web site content and performance through multivariate testing*, https://info.sitespect.com/hubfs/Case%20Studies/Mozilla%20x%20SiteSpect_Case%20Study.pdf, [Accessed 04-01-2026], 2011.

[12] P. Caserman, M. Martinussen, and S. Göbel, "Effects of end-to-end latency on user experience and performance in immersive virtual reality applications," in *Entertainment Computing and Serious Games: First IFIP TC 14 Joint International Conference, ICEC-JCSG 2019, Arequipa, Peru, November 11–15, 2019, Proceedings*, Arequipa, Peru: Springer-Verlag, 2019, 57–69, ISBN: 978-3-030-34643-0. DOI: 10.1007/978-3-030-34644-7_5. [Online]. Available: https://doi.org/10.1007/978-3-030-34644-7_5.

[13] D. Monaco et al., "Real-time latency prediction for cloud gaming applications," *Comput. Netw.*, vol. 264, no. C, Jun. 2025, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2025.111235. [Online]. Available: https://doi.org/10.1016/j.comnet.2025.111235.

[14] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, ser. SIGIR '14, Gold Coast, Queensland, Australia: Association for Computing Machinery, 2014, 103–112, ISBN: 9781450322577. DOI: 10.1145/2600428.2609627. [Online]. Available: https://doi.org/10.1145/2600428.2609627.

[15] Z. Wang, W. Liu, J. Chen, and T.-H. P. Chen, "Rperf: Mining user reviews using topic modeling to assist performance testing: An industrial experience report," in *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '25, Toronto ON, Canada: Association for Computing Machinery, 2025, p. 1, ISBN: 9798400711305. DOI: 10.1145/3680256.3721264. [Online]. Available: https://doi.org/10.1145/3680256.3721264.

[16] C. U. Smith, "Software performance engineering," in *Performance Evaluation of Computer and Communication Systems*, L. Donatiello and R. Nelson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–536, ISBN: 978-3-540-48044-0.

[17] C. U. Smith and L. G. Williams, "Software performance engineering," in *UML for Real: Design of Embedded Real-Time Systems*, L. Lavagno, G. Martin, and B. Selic, Eds. Boston, MA: Springer US, 2003, pp. 343–365, ISBN: 978-0-306-48738-5. DOI: 10.1007/0-306-48738-1_16. [Online]. Available: https://doi.org/10.1007/0-306-48738-1_16.

[18] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*, ser. FOSE '07, USA: IEEE Computer Society, 2007, 171–187, ISBN: 0769528295. DOI: 10.1109/FOSE.2007.32. [Online]. Available: https://doi.org/10.1109/FOSE.2007.32.

[19] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer Berlin Heidelberg, 2011, ISBN: 9783642136214. DOI: 10.1007/978-3-642-13621-4. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13621-4.

[20] A. B. Bondi, "Incorporating software performance engineering methods and practices into the software development life cycle," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16, Delft, The Netherlands: Association for Computing Machinery, 2016, 327–330, ISBN: 9781450340809. DOI: 10.1145/2851553.2858668. [Online]. Available: https://doi.org/10.1145/2851553.2858668.

[21] R. Eramo, M. Tucci, D. D. Pompeo, V. Cortellessa, A. D. Marco, and D. Taibi, "Architectural support for software performance in continuous software engineering: A systematic mapping study," *J. Syst. Softw.*, vol. 207, p. 111 833, 2024. DOI: 10.1016/J.JSS.2023.111833. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111833.

[22] L. Liao et al., "Early detection of performance regressions by bridging local performance data and architectural models," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, IEEE, 2025, pp. 2841–2853. DOI: 10.1109/ICSE55347.2025.00026. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00026.

[23] P. C. Brebner, "A performance modeling "blending" approach for early lifecycle risk mitigation," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12, Boston, Massachusetts, USA: Association for Computing Machinery, 2012, 271–274, ISBN: 9781450312028. DOI: 10.1145/2188286.2188336. [Online]. Available: https://doi.org/10.1145/2188286.2188336.

[24] M. Menninghaus and E. Pulvermüller, "Towards using code coverage metrics for performance comparison on the implementation level," in *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*, A. Avritzer, A. Iosup, X. Zhu, and S. Becker, Eds., ACM, 2016, pp. 101–104. DOI: 10.1145/2851553.2858663. [Online]. Available: https://doi.org/10.1145/2851553.2858663.

[25] I. Molyneaux, *The Art of Application Performance Testing*. O'Reilly, 2015, ISBN: 978-1-49190-054-3.

[26] D. A. Menascé, "Load testing of web sites," *IEEE Internet Computing*, vol. 6, no. 4, 70–74, Jul. 2002, ISSN: 1089-7801. DOI: 10.1109/MIC.2002.1020328. [Online]. Available: https://doi.org/10.1109/MIC.2002.1020328.

[27] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *Proceedings of the 3rd International Workshop on Software and Performance*, ser. WOSP '02, Rome, Italy: Association for Computing Machinery, 2002, 17–24, ISBN: 1581135637. DOI: 10.1145/584369.584373. [Online]. Available: https://doi.org/10.1145/584369.584373.

[28] D. Daly, "Creating a virtuous cycle in performance testing at mongodb," in *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021*, J. Bourcier, Z. M. J. Jiang, C. Bezemer, V. Cortellessa, D. D. Pompeo, and A. L. Varbanescu, Eds., ACM, 2021, pp. 33–41. DOI: 10.1145/3427921.3450234. [Online]. Available: https://doi.org/10.1145/3427921.3450234.

[29] A. Bauer, M. Züfle, N. Herbst, A. Zehe, A. Hotho, and S. Kounev, "Time Series Forecasting for Self-Aware Systems," *Proceedings of the IEEE*, vol. 108, no. 7, pp. 1068–1093, 2020. DOI: 10.1109/JPROC.2020.2983857.

[30] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 1–12.

[31] L. Klaver, T. van der Knaap, J. van der Geest, E. Harmsma, B. van der Waaij, and P. Pileggi, "Towards independent run-time cloud monitoring," in *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, Companion Volume*, J. Bourcier, Z. M. J. Jiang, C. Bezemer, V. Cortellessa, D. D. Pompeo, and A. L. Varbanescu, Eds., ACM, 2021, pp. 21–26. DOI: 10.1145/3447545.3451180. [Online]. Available: https://doi.org/10.1145/3447545.3451180.

[32] S. Faasse, J. Bucek, and D. Schmidt, "Out of band performance monitoring of server workloads: Leveraging restful API to monitor compute resource utilization and performance related metrics for server performance analysis," in *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, J. N. Amaral, A. Koziolek, C. Trubiani, and A. Iosup, Eds., ACM, 2020, pp. 4–11. DOI: 10.1145/3358960.3375795. [Online]. Available: https://doi.org/10.1145/3358960.3375795.

[33] L. Traini, "Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study," *Empirical Software Engineering*, vol. 27, no. 3, p. 74, Mar. 2022, ISBN: 1573-7616. DOI: 10.1007/s10664-021-10069-3. [Online]. Available: https://doi.org/10.1007/s10664-021-10069-3.

[34] P. Brebner, "Real-world performance modelling of enterprise service oriented architectures: Delivering business value with complexity and constraints," in *ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011*, S. Kounev, V. Cortellessa, R. Mirandola, and D. J. Lilja, Eds., ACM, 2011, pp. 85–96. DOI: 10.1145/1958746.1958762. [Online]. Available: https://doi.org/10.1145/1958746.1958762.

[35] D. D. Gouvêa et al., "Experience building non-functional requirement models of a complex industrial architecture," in *ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011*, S. Kounev, V. Cortellessa, R. Mirandola, and D. J. Lilja, Eds., ACM, 2011, pp. 43–54. DOI: 10.1145/1958746.1958757. [Online]. Available: https://doi.org/10.1145/1958746.1958757.

[36] A. Ilyushkin et al., "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, Eds., ACM, 2017, pp. 75–86. DOI: 10.1145/3030207.3030214. [Online]. Available: https://doi.org/10.1145/3030207.3030214.

[37] H. Koziolek, "Performance engineering practices for modern industrial applications at abb research," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '23 Companion, Coimbra, Portugal: Association for Computing Machinery, 2023, p. 339, ISBN: 9798400700729. DOI: 10.1145/3578245.3584350. [Online]. Available: https://doi.org/10.1145/3578245.3584350.

[38] A. Ezaz, G. Khodabandeh, and N. Ezzati-Jivan, "Analyzing performance variability in alibaba's microservice architecture: A critical-path-based perspective," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '24 Companion, London, United Kingdom: Association for Computing Machinery, 2024, 82–86, ISBN: 9798400704451. DOI: 10.1145/3629527.3651845. [Online]. Available: https://doi.org/10.1145/3629527.3651845.

[39] A. Busch and M. Kammerer, "Network performance influences of software-defined networks on micro-service architectures," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21, Virtual Event, France: Association for Computing Machinery, 2021, 153–163, ISBN: 9781450381949. DOI: 10.1145/3427921.3450236. [Online]. Available: https://doi.org/10.1145/3427921.3450236.

[40] D. Daly, W. Brown, H. Ingo, J. O'Leary, and D. Bradford, "The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, event-place: Edmonton AB, Canada, New York, NY, USA: Association for Computing Machinery, 2020, pp. 67–75, ISBN: 978-1-4503-6991-6. DOI: 10.1145/3358960.3375791. [Online]. Available: https://doi.org/10.1145/3358960.3375791.

[41] J. Rubin and M. Rinard, "The Challenges of Staying Together While Moving Fast: An Exploratory Study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, event-place: Austin, Texas, New York, NY, USA: Association for Computing Machinery, 2016, pp. 982–993, ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884871. [Online]. Available: https://doi.org/10.1145/2884781.2884871.

[42] L. Traini et al., "How Software Refactoring Impacts Execution Time," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, Dec. 2021, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 1049-331X. DOI: 10.1145/3485136. [Online]. Available: https://doi.org/10.1145/3485136.

[43] M. Di Penta, G. Bavota, and F. Zampetti, "On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, event-place: Virtual Event, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 556–567, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409695. [Online]. Available: https://doi.org/10.1145/3368089.3409695.

[44] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice," *IEEE Software*, vol. 30, no. 5, pp. 30–37, 2013. DOI: 10.1109/MS.2013.94.

[45] 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, event-place: Salerno, Italy, New York, NY, USA: Association for Computing Machinery, 2024, pp. 48–57, ISBN: 979-8-4007-1701-7. DOI: 10.1145/3661167.3661196. [Online]. Available: https://doi.org/10.1145/3661167.3661196.

[46] M. Imran, V. Cortellessa, D. D. Ruscio, R. Rubei, and L. Traini, "Is code coverage of performance tests related to source code features? an empirical study on open-source java systems," *Empir. Softw. Eng.*, vol. 30, no. 6, p. 157, 2025. DOI: 10.1007/S10664-025-10712-3. [Online]. Available: https://doi.org/10.1007/s10664-025-10712-3.

[47] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?" In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, event-place: Seoul, South Korea, New York, NY, USA: Association for Computing Machinery, 2020, pp. 1435–1446, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380351. [Online]. Available: https://doi.org/10.1145/3377811.3380351.

[48] T. Makkouk, D. J. Kim, and T.-H. P. Chen, "An empirical study on performance bugs in deep learning frameworks," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 35–46. DOI: 10.1109/ICSME55016.2022.00012.

[49] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 199–208. DOI: 10.1109/MSR.2012.6224281.

[50] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *SIGPLAN Not.*, vol. 47, no. 6, 77–88, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254075. [Online]. Available: https://doi.org/10.1145/2345156.2254075.

[51] Y. Zhao, L. Xiao, A. B. Bondi, B. Chen, and Y. Liu, "A large-scale empirical study of real-life performance issues in open source projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 924–946, 2023. DOI: 10.1109/TSE.2022.3167628.

[52] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding," ser. KDD '18, event-place: London, United Kingdom, New York, NY, USA: Association for Computing Machinery, 2018, pp. 387–395, ISBN: 978-1-4503-5552-0. DOI: 10.1145/3219819.3219845. [Online]. Available: https://doi.org/10.1145/3219819.3219845.

[53] K. Jia, X. Yu, C. Zhang, W. Hu, D. Zhao, and J. Xiang, "Software Aging Prediction for Cloud Services Using a Gate Recurrent Unit Neural Network Model Based on Time Series Decomposition," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 3, pp. 580–593, 2023. DOI: 10.1109/TETC.2023.3258503.

[54] Y. Syu, J.-Y. Kuo, and Y.-Y. Fanjiang, "Time series forecasting for dynamic quality of web services: An empirical study," *Journal of Systems and Software*, vol. 134, pp. 279–303, 2017, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.09.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217302042.

[55] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin, "Predicting Performance Anomalies in Software Systems at Run-Time," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, Apr. 2021, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 1049-331X. DOI: 10.1145/3440757. [Online]. Available: https://doi.org/10.1145/3440757.

[56] V. R. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana, and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling Web applications hosted in the cloud infrastructure," *Neural Computing and Applications*, vol. 27, no. 8, pp. 2383–2406, Nov. 2016, ISBN: 1433-3058. DOI: 10.1007/s00521-015-2133-3. [Online]. Available: https://doi.org/10.1007/s00521-015-2133-3.

[57] L. Traini and V. Cortellessa, "DeLag: Using Multi-Objective Optimization to Enhance the Detection of Latency Degradation Patterns in Service-Based Systems," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3554–3580, 2023. DOI: 10.1109/TSE.2023.3266041.

[58] A. Baluta, Y. Rouf, J. Mukherjee, Z. M. Jiang, and M. Litoiu, "Disambiguating performance anomalies from workload changes in cloud-native applications," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '24, London, United Kingdom: Association for Computing Machinery, 2024, 286–297, ISBN: 9798400704444. DOI: 10.1145/3629526.3645046. [Online]. Available: https://doi.org/10.1145/3629526.3645046.

[59] S. He, B. Yang, and Q. Qiao, "Overview of Key Performance Indicator Anomaly Detection," in *2021 IEEE Region 10 Symposium (TENSYMP)*, 2021, pp. 1–6. DOI: 10.1109/TENSYMP52854.2021.9550989.

[60] J. Zhou, Y. Qian, Q. Zou, P. Liu, and J. Xiang, "DeepSyslog: Deep Anomaly Detection on Syslog Using Sentence Embedding and Metadata," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3051–3061, 2022. DOI: 10.1109/TIFS.2022.3201379.

[61] C. Lee, T. Yang, Z. Chen, Y. Su, Y. Yang, and M. R. Lyu, "Heterogeneous Anomaly Detection for Software Systems via Semi-supervised Cross-modal Attention," *ArXiv*, vol. abs/2302.06914, 2023.

[62] N. Zhao et al., "An Empirical Investigation of Practical Log Anomaly Detection for Online Service Systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, event-place: Athens, Greece, New York, NY, USA: Association for Computing Machinery, 2021, pp. 1404–1415, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3473933. [Online]. Available: https://doi.org/10.1145/3468264.3473933.

[63] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, "Linkster: Enabling efficient manual inspection and annotation of mined data," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, Santa Fe, New Mexico, USA: Association for

Computing Machinery, 2010, 369–370, ISBN: 9781605587912. DOI: 10.1145/1882291.1882352. [Online]. Available: https://doi.org/10.1145/1882291.1882352.

[64] D. Zhang, Y. Dang, J.-G. Lou, S. Han, H. Zhang, and T. Xie, "Software analytics as a learning case in practice: Approaches and experiences," in *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, ser. MALETS '11, Lawrence, Kansas, USA: Association for Computing Machinery, 2011, 55–58, ISBN: 9781450310222. DOI: 10.1145/2070821.2070829. [Online]. Available: https://doi.org/10.1145/2070821.2070829.

[65] H. K. Dam, T. Tran, and A. Ghose, "Explainable software analytics," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, 53–56, ISBN: 9781450356626. DOI: 10.1145/3183399.3183424. [Online]. Available: https://doi.org/10.1145/3183399.3183424.

[66] A. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008.

[67] W. Sun et al., "Source Code Summarization in the Era of Large Language Models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 419–431. DOI: 10.1109/ICSE55347.2025.00034. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00034.

[68] T. Ahmed, K. S. Pai, P. T. Devanbu, and E. T. Barr, "Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, 220:1–220:13. DOI: 10.1145/3597503.3639183. [Online]. Available: https://doi.org/10.1145/3597503.3639183.

[69] K. Z. Sultana, "Towards a software vulnerability prediction model using traceable code patterns and software metrics," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 1022–1025. DOI: 10.1109/ASE.2017.8115724.

[70] Y. Ding, B. Steenhoek, K. Pei, G. Kaiser, W. Le, and B. Ray, "TRACED: Execution-aware Pre-training for Source Code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, event-place: Lisbon, Portugal, New York, NY, USA: Association for Computing Machinery, 2024, ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3608140. [Online]. Available: https://doi.org/10.1145/3597503.3608140.

[71] A. Sejfia, S. Das, S. Shafiq, and N. Medvidovic, "Toward Improved Deep Learning-based Vulnerability Detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, 62:1–62:12. DOI: 10.1145/3597503.3608141. [Online]. Available: https://doi.org/10.1145/3597503.3608141.

[72] M. Ciniselli, L. Pascarella, and G. Bavota, "Deep learning-based code completion: On the impact on performance of contextual information," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 211–223. DOI: 10.1109/ICSME58944.2024.00029.

[73] D. Kim, "Comparative analysis of design pattern implementation validity in llm-based code refactoring," *J. Syst. Softw.*, vol. 230, p. 112 519, 2025. DOI: 10.1016/J.JSS.2025.112519. [Online]. Available: https://doi.org/10.1016/j.jss.2025.112519.

[74] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015. DOI: 10.1109/TCC.2014.2350475.

[75] J. Kumar and A. K. Singh, "Performance Assessment of Time Series Forecasting Models for Cloud Datacenter Networks'Workload Prediction," *Wireless Personal Communications*, vol. 116, no. 3, pp. 1949–1969, Feb. 2021, ISBN: 1572-834X. DOI: 10.1007/s11277-020-07773-6. [Online]. Available: https://doi.org/10.1007/s11277-020-07773-6.

[76] X. Chen et al., "Hetfl: Heterogeneous graph-based software fault localization," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2884–2905, 2024. DOI: 10.1109/TSE.2024.3454605.

[77] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, 169–180, ISBN: 9781450362245. DOI: 10.1145/3293882.3330574. [Online]. Available: https://doi.org/10.1145/3293882.3330574.

[78] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Santa Barbara, CA, USA: Association for Computing Machinery, 2017, 273–283, ISBN: 9781450350761. DOI: 10.1145/3092703.3092717. [Online]. Available: https://doi.org/10.1145/3092703.3092717.

[79] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "DeepDevPERF: A deep learning-based approach for improving software performance," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, event-place: Singapore, Singapore, New York, NY, USA: Association for Computing Machinery, 2022, pp. 948–958, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549096. [Online]. Available: https://doi.org/10.1145/3540250.3549096.

[80] Z. Chen, S. Fang, and M. Monperrus, "SUPERSONIC: Learning to Generate Source Code Optimizations in C/C++," *IEEE Transactions on Software Engineering*, pp. 1–17, 2024. DOI: 10.1109/TSE.2024.3423769.

[81] X. Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 1049-331X. DOI: 10.1145/3695988. [Online]. Available: https://doi.org/10.1145/3695988.

[82] Y. Ding et al., "Vulnerability Detection with Code Language Models: How Far Are We?" In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481. DOI: 10.1109/ICSE55347.2025.00038. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00038.

[83] Y. Xiao, V.-H. Le, and H. Zhang, "Demonstration-Free: Towards More Practical Log Parsing with Large Language Models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, event-place: Sacramento, CA, USA, New York, NY, USA: Association for Computing Machinery, 2024, pp. 153–165, ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3694994. [Online]. Available: https://doi.org/10.1145/3691620.3694994.

[84] A. Shypula et al., "Learning Performance-Improving Code Edits," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=ix7rLVHXyY.

[85] S. Gao, C. Gao, W. Gu, and M. Lyu, "Search-Based LLMs for Code Optimization," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 254–266. DOI: 10.1109/ICSE55347.2025.00021. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00021.

[86] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "RAPGen: An Approach for Fixing Code Inefficiencies in Zero-Shot," *CoRR*, vol. abs/2306.17077, 2023, arXiv: 2306.17077. DOI: 10.48550/ARXIV.2306.17077. [Online]. Available: https://doi.org/10.48550/arXiv.2306.17077.

[87] W. Ma et al., *Lms: Understanding code syntax and semantics for code analysis*, 2023. DOI: 10.48550/ARXIV.2305.12138. [Online]. Available: https://arxiv.org/abs/2305.12138.

[88] A. Ni et al., "NExT: Teaching Large Language Models to Reason about Code Execution," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=B1W712hMBi.

[89] Y. Ding, J. Peng, M. J. Min, G. E. Kaiser, J. Yang, and B. Ray, "SemCoder: Training Code Language Models with Comprehensive Semantics," *CoRR*, vol. abs/2406.01006, 2024, arXiv: 2406.01006. DOI: 10.48550/ARXIV.2406.01006. [Online]. Available: https://doi.org/10.48550/arXiv.2406.01006.

[90] P. Berube, A. Preuss, and J. N. Amaral, "Combined profiling: Practical collection of feedback information for code optimization," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11, Karlsruhe, Germany: Association for Computing Machinery, 2011, 493–498, ISBN: 9781450305198. DOI: 10.1145/1958746.1958821. [Online]. Available: https://doi.org/10.1145/1958746.1958821.

[91] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open Code Large Language Models for Code Understanding and Generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, H. Bouamor,

J. Pino, and K. Bali, Eds., Association for Computational Linguistics, 2023, pp. 1069–1088. DOI: 10.18653/V1/2023.EMNLP-MAIN.68. [Online]. Available: https://doi.org/10.18653/v1/2023.emnlp-main.68.

[92] L. B. R. dos Santos, E. F. de Souza, A. T. Endo, C. Trubiani, R. Pinciroli, and N. L. Vijaykumar, "Performance regression testing initiatives: A systematic mapping," *Inf. Softw. Technol.*, vol. 179, no. C, Mar. 2025, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2024.107641. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107641.

[93] L. Liao et al., "Locating performance regression root causes in the field operations of web-based systems: An experience report," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4986–5006, 2022. DOI: 10.1109/TSE.2021.3131529.

[94] 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, event-place: L'Aquila, Italy, New York, NY, USA: Association for Computing Machinery, 2017, pp. 373–384, ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030213. [Online]. Available: https://doi.org/10.1145/3030207.3030213.

[95] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming Performance Variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 409–425, ISBN: 978-1-939133-08-3. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq.

[96] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically sound performance evaluation," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, event-place: Houston, Texas, USA, New York, NY, USA: Association for Computing Machinery, 2013, pp. 219–228, ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451141. [Online]. Available: https://doi.org/10.1145/2451116.2451141.

[97] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, event-place: Washington, DC, USA, New York, NY, USA: Association for Computing Machinery, 2009, pp. 265–276, ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508275. [Online]. Available: https://doi.org/10.1145/1508244.1508275.

[98] T. Kalibera and R. Jones, "Rigorous Benchmarking in Reasonable Time," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13, event-place: Seattle, Washington, USA, New York, NY, USA: Association for Computing Machinery, 2013, pp. 63–74, ISBN: 978-1-4503-2100-6. DOI: 10.1145/2464157.2464160. [Online]. Available: https://doi.org/10.1145/2464157.2464160.

[99] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An Automated Approach for Recommending When to Stop Performance Tests," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 279–289. DOI: 10.1109/ICSME.2016.46.

[100] T.-H. Chen et al., "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)*, 2017, pp. 243–252. DOI: 10.1109/ICSE-SEIP.2017.26.

[101] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa, "A Statistics-Based Performance Testing Methodology for Cloud Applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, event-place: Tallinn, Estonia, New York, NY, USA: Association for Computing Machinery, 2019, pp. 188–199, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338912. [Online]. Available: https://doi.org/10.1145/3338906.3338912.

[102] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "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*, ser. ESEC/FSE 2020, event-place: Virtual Event, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 989–1001, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409683. [Online]. Available: https://doi.org/10.1145/3368089.3409683.

[103] H. M. AlGhamdi, C.-P. Bezemer, W. Shang, A. E. Hassan, and P. Flora, "Towards reducing the time needed for load testing," *Journal of Software: Evolution and Process*, vol. 35, no. 3, e2276, 2023. DOI: https://doi.org/10.1002/smr.2276. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2276.

[104] H. Jia et al., "Detecting jvm jit compiler bugs via exploring two-dimensional input spaces," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, 43–55, ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00016. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00016.

[105] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of jit/jvm optimizations on java application performance," in *Seventh Workshop on Interaction Between Compilers and Computer Architectures, 2003. INTERACT-7 2003. Proceedings.*, 2003, pp. 5–13. DOI: 10.1109/INTERA.2003.1192351.

[106] A. Prokopec et al., "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, 31–47, ISBN: 9781450367127. DOI: 10.1145/3314221.3314637. [Online]. Available: https://doi.org/10.1145/3314221.3314637.

[107] L. Traini, V. Cortellessa, D. Di Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in Java software: Are we there yet?" *Empirical Software Engineering*, vol. 28, no. 1, p. 13, Nov. 2022, ISBN: 1573-7616. DOI: 10.1007/s10664-022-10247-x. [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x.

[108] 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, Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3133876. [Online]. Available: https://doi.org/10.1145/3133876.

[109] 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, event-place: Montreal, Quebec, Canada, New York, NY, USA: Association for Computing Machinery, 2007, pp. 57–76, ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. [Online]. Available: https://doi.org/10.1145/1297027.1297033.

[110] OpenJDK, *Java microbenchmarking harness (jmh)*, https://github.com/openjdk/jmh/, Accessed: 2024-09-09, 2014.

[111] K. Veeraraghavan et al., "Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 635–651, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/veeraraghavan.

[112] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? challenges for production-ready autoscaling," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22, Beijing, China: Association for Computing Machinery, 2022, 105–115, ISBN: 9781450391436. DOI: 10.1145/3489525.3511680. [Online]. Available: https://doi.org/10.1145/3489525.3511680.

[113] M. Shahrad et al., "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 205–218, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad.

[114] J. Boudjadar and S. Nadjm-Tehrani, "Schedulability and memory interference analysis of multicore preemptive real-time systems," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, Eds., ACM, 2017, pp. 263–274. DOI: 10.1145/3030207.3030233. [Online]. Available: https://doi.org/10.1145/3030207.3030233.

[115] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, Jul. 2015, ISSN: 0360-0300. DOI: 10.1145/2791577. [Online]. Available: https://doi.org/10.1145/2791577.

[116] *Why Appinventiv Releases Software Multiple Times A Day? — appinventiv.com*, https://appinventiv.com/blog/release-software-to-production/, [Accessed 03-12-2025].

[117] *Why Great Product Companies Release Software to Production Multiple Times a Day — dzone.com*, https://dzone.com/articles/why-do-great-product-companies-release-software-to, [Accessed 03-12-2025].

[118] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, 169–180, Aug. 2002, ISSN: 1089-7089. DOI: 10.1002/bltj.2229. [Online]. Available: http://dx.doi.org/10.1002/bltj.2229.

[119] A. Gartziandia, "Microservice-Based Performance Problem Detection in Cyber-Physical System Software Updates," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 147–149. DOI: 10.1109/ICSE-Companion52605.2021.00062.

[120] Z. Li et al., "Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 389–402, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/li.

[121] S. Zhang et al., "Rapid and Robust Impact Assessment of Software Changes in Large Internet-Based Services," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15, event-place: Heidelberg, Germany, New York, NY, USA: Association for Computing Machinery, 2015, ISBN: 978-1-4503-3412-9. DOI: 10.1145/2716281.2836087. [Online]. Available: https://doi.org/10.1145/2716281.2836087.

[122] D. Zagieboylo and K. A. Zaman, "Cost-efficient and reliable reporting of highly bursty video game crash data," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, L'Aquila, Italy: Association for Computing Machinery, 2017, 201–212, ISBN: 9781450344043. DOI: 10.1145/3030207.3044529. [Online]. Available: https://doi.org/10.1145/3030207.3044529.

[123] S. W. Shi, S. Lee, K. Kalyanam, and M. Wedel, "The impact of app crashes on consumer engagement," *Journal of Marketing*, vol. 89, no. 4, 81–98, Mar. 2025, ISSN: 1547-7185. DOI: 10.1177/00222429241304322. [Online]. Available: http://dx.doi.org/10.1177/00222429241304322.

[124] *What Mobile Users Expect from Mobile Apps - Applause — applause.com*, https://www.applause.com/blog/what-mobile-users-expect-from-mobile-apps/, [Accessed 03-12-2025].

[125] T. Su et al., "Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1115–1137, 2022. DOI: 10.1109/TSE.2020.3013438.

[126] A. Schörgenhumer, M. Kahlhofer, H. Mössenböck, and P. Grünbacher, "Using crash frequency analysis to identify error-prone software technologies in multi-system monitoring," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 183–190. DOI: 10.1109/QRS.2018.00032.

[127] L. Fong-Jones and C. Majors, *Observability Engineering*. O'Reilly Media, Incorporated, 2022.

[128] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.

[129] A. Amin, L. Grunske, and A. Colman, "An approach to software reliability prediction based on time series modeling," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1923–1932, 2013, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.03.045. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000617.

[130] H. Lu, W. Kolarik, and S. Lu, "Real-time performance reliability prediction," *IEEE Transactions on Reliability*, vol. 50, no. 4, 2001. DOI: 10.1109/24.983393.

[131] D. Ardelean, A. Diwan, and C. Erdman, "Performance Analysis of Cloud Applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 405–417, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/ardelean.

[132] A. F. Ansari et al., "Chronos: Learning the Language of Time Series," *Transactions on Machine Learning Research*, 2024, ISSN: 2835-8856. [Online]. Available: https://openreview.net/forum?id=gerNCVqqtR.

[133] A. Das, W. Kong, R. Sen, and Y. Zhou, "A decoder-only foundation model for time-series forecasting," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=jn2iTJas6h.

[134] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997. DOI: 10.1109/4235.585893.

[135] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, "What is the Connection between Issues, Bugs, and Enhancements? Lessons Learned from 800+ Software Projects," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18, event-place: Gothenburg, Sweden, New York, NY, USA: Association for Computing Machinery, 2018, pp. 306–315, ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183548. [Online]. Available: https://doi.org/10.1145/3183519.3183548.

[136] M. Robredo, N. Saarimaki, M. Esposito, D. Taibi, R. Penaloza, and V. Lenarduzzi, *Evaluating Time-Dependent Methods and Seasonal Effects in Code Technical Debt Prediction*, _eprint: 2408.08095, 2025.

[137] R. B. Prudêncio and T. B. Ludermir, "Meta-learning approaches to selecting time series models," *Neurocomputing*, vol. 61, pp. 121–137, Oct. 2004, Publisher: Elsevier BV, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2004.03.008. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2004.03.008.

[138] S. Shahoud, H. Khalloof, C. Duepmeier, and V. Hagenmeyer, "Descriptive Statistics Time-based Meta Features (DSTMF): Constructing a better Set of Meta Features for Model Selection in Energy Time Series Forecasting," in *Proceedings of the 3rd International Conference on Applications of Intelligent Systems*, ser. APPIS 2020, ACM, 2020, ISBN: 978-1-4503-7630-3. DOI: 10.1145/3378184.3378221. [Online]. Available: https://doi.org/10.1145/3378184.3378221.

[139] T. S. Talagala, R. J. Hyndman, and G. Athanasopoulos, "Meta-learning how to forecast time series," *Journal of Forecasting*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:70072190.

[140] K. A. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Computing Surveys*, vol. 41, no. 1, pp. 1–25, Jan. 2009, ISSN: 1557-7341. DOI: 10.1145/1456650.1456656. [Online]. Available: http://dx.doi.org/10.1145/1456650.1456656.

[141] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013. DOI: 10.1109/TSE.2012.64.

[142] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A Survey on Compiler Autotuning using Machine Learning," *ACM Comput. Surv.*, vol. 51, no. 5, Sep. 2018, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0360-0300. DOI: 10.1145/3197978. [Online]. Available: https://doi.org/10.1145/3197978.

[143] J. Huang, J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen, "Code Representation Pre-training with Complements from Program Executions," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, F. Dernoncourt, D. Preoţiuc-Pietro, and A. Shimorina, Eds., Miami, Florida, US: Association for Computational Linguistics, Nov. 2024, pp. 267–278. [Online]. Available: https://aclanthology.org/2024.emnlp-industry.21.

[144] B. Souza and M. Pradel, "LExecutor: Learning-Guided Execution," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, event-place: San Francisco, CA, USA, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1522–1534, ISBN: 979-8-4007-0327-0. DOI: 10.1145/3611643.3616254. [Online]. Available: https://doi.org/10.1145/3611643.3616254.

[145] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.*, 2004, pp. 36–43. DOI: 10.1109/ICAC.2004.1301345.

[146] A. Mastropaolo, F. Zampetti, G. Bavota, and M. D. Penta, "Toward Automatically Completing GitHub Workflows," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, 13:1–13:12. DOI: 10.1145/3597503.3623351. [Online]. Available: https://doi.org/10.1145/3597503.3623351.

[147] A. Mastropaolo, M. Ciniselli, L. Pascarella, R. Tufano, E. Aghajani, and G. Bavota, "Towards Summarizing Code Snippets Using Pre-Trained Transformers," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15-16, 2024*, I. Steinmacher, M. Linares-Vásquez, K. P. Moran, and O. Baysal, Eds., ACM, 2024, pp. 1–12. DOI: 10.1145/3643916.3644400. [Online]. Available: https://doi.org/10.1145/3643916.3644400.

[148] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, "Knowledge-Aware Code Generation with Large Language Models," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15-16, 2024*, I. Steinmacher, M. Linares-Vásquez, K. P. Moran, and O. Baysal, Eds., ACM, 2024, pp. 52–63. DOI: 10.1145/3643916.3644418. [Online]. Available: https://doi.org/10.1145/3643916.3644418.

[149] A. D. Gotmare, J. Li, S. Joty, and S. C. H. Hoi, "Efficient Text-to-Code Retrieval with Cascaded Fast and Slow Transformer Models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds., ACM, 2023, pp. 388–400. DOI: 10.1145/3611643.3616369. [Online]. Available: https://doi.org/10.1145/3611643.3616369.

[150] L. Song and S. Lu, "Performance Diagnosis for Inefficient Loops," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 370–380. DOI: 10.1109/ICSE.2017.41.

[151] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 902–912. DOI: 10.1109/ICSE.2015.100.

[152] R. Puri et al., "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S.-K. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html.

[153] Z. Luo et al., "WizardCoder: Empowering Code Large Language Models with Evol-Instruct," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=UnUwSIgK5W.

[154] H. Yu et al., "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, 37:1–37:12. DOI: 10.1145/3597503.3623316. [Online]. Available: https://doi.org/10.1145/3597503.3623316.

[155] A. Mastropaolo, V. Nardone, G. Bavota, and M. D. Penta, "How the Training Procedure Impacts the Performance of Deep Learning-based Vulnerability Patching," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024*, ACM, 2024, pp. 150–159. DOI: 10.1145/3661167.3661200. [Online]. Available: https://doi.org/10.1145/3661167.3661200.

[156] N. L. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[157] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, Publisher: [International Biometric Society, Wiley], ISSN: 00994987. Accessed: May 29, 2024. [Online]. Available: http://www.jstor.org/stable/3001968.

[158] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. DOI: 10.3102/10769986025002101. [Online]. Available: https://doi.org/10.3102/10769986025002101.

[159] D. S. Kerby, "The Simple Difference Formula: An Approach to Teaching Nonparametric Correlation," *Comprehensive Psychology*, vol. 3, 11.IT.3.1, 2014, _eprint: https://doi.org/10.2466/11.IT.3.1. DOI: 10.2466/11.IT.3.1. [Online]. Available: https://doi.org/10.2466/11.IT.3.1.

[160] Z. M. Jiang and A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015. DOI: 10.1109/TSE.2015.2445340.

[161] M. Fagerström et al., "Verdict Machinery: On the Need to Automatically Make Sense of Test Results," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 225–234, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931064. [Online]. Available: https://doi.org/10.1145/2931037.2931064.

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

[163] B. Everitt and A. Skrondal, *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2010, ISBN: 978-0-521-76699-9.

[164] A. C. Davison and D. V. Hinkley, *Bootstrap methods and their application*. Cambridge University Press, 1997.

[165] T. Kalibera and R. Jones, "Quantifying Performance Changes with Effect Size Confidence Intervals," University of Kent, Technical Report 4–12, Jun. 2012, p. 55. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2012/3233.

[166] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79 –86, 1951, Publisher: Institute of Mathematical Statistics. DOI: 10.1214/aoms/1177729694. [Online]. Available: https://doi.org/10.1214/aoms/1177729694.

[167] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu, and Y. Liu, "Deep Learning for Sensor-based Human Activity Recognition: Overview, Challenges, and Opportunities," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0360-0300. DOI: 10.1145/3447744. [Online]. Available: https://doi.org/10.1145/3447744.

[168] A. Rajkomar et al., "Scalable and accurate deep learning for electronic health records," *npj Digital Medicine*, vol. 1, Jan. 2018. DOI: 10.1038/s41746-018-0029-1.

[169] M. Arul and A. Kareem, "Applications of shapelet transform to time series classification of earthquake, wind and wave data," *Engineering Structures*, vol. 228, p. 111 564, 2021, ISSN: 0141-0296. DOI: https://doi.org/10.1016/j.engstruct.2020.111564. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141029620341651.

[170] S. Majumdar and A. K. Laha, "Clustering and classification of time series using topological data analysis with applications to finance," *Expert Systems with Applications*, vol. 162, p. 113 868, 2020, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113868. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742030676X.

[171] M. Middlehurst, P. Schäfer, and A. J. Bagnall, "Bake off redux: A review and experimental evaluation of recent time series classification algorithms," *CoRR*, vol. abs/2304.13029, 2023, arXiv: 2304.13029. DOI: 10.48550/ARXIV.2304.13029. [Online]. Available: https://doi.org/10.48550/arXiv.2304.13029.

[172] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, "The great time series classification bake off: A review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 31, no. 3, pp. 606–660, May 2017, ISBN: 1573-756X. DOI: 10.1007/s10618-016-0483-9. [Online]. Available: https://doi.org/10.1007/s10618-016-0483-9.

[173] J. Lines and A. Bagnall, "Time series classification with ensembles of elastic distance measures," *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 565–592, May 2015, ISSN: 1573-756X. DOI: 10.1007/s10618-014-0361-2. [Online]. Available: https://doi.org/10.1007/s10618-014-0361-2.

[174] P. Schäfer, "The BOSS is concerned with time series classification in the presence of noise," *Data Mining and Knowledge Discovery*, vol. 29, no. 6, pp. 1505–1530, Nov. 2015, ISBN: 1573-756X. DOI: 10.1007/s10618-014-0377-7. [Online]. Available: https://doi.org/10.1007/s10618-014-0377-7.

[175] C. W. Tan, A. Dempster, C. Bergmeir, and G. I. Webb, "MultiRocket: Multiple pooling operators and transformations for fast and effective time series classification," *Data Min. Knowl. Discov.*, vol. 36, no. 5, pp. 1623–1646, Sep. 2022, Place: USA Publisher: Kluwer Academic Publishers, ISSN: 1384-5810. DOI: 10.1007/s10618-022-00844-1. [Online]. Available: https://doi.org/10.1007/s10618-022-00844-1.

[176] 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, Sep. 2020, ISBN: 1573-756X. DOI: 10.1007/s10618-020-00701-z. [Online]. Available: https://doi.org/10.1007/s10618-020-00701-z.

[177] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: A review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, Jul. 2019, ISBN: 1573-756X. DOI: 10.1007/s10618-019-00619-1. [Online]. Available: https://doi.org/10.1007/s10618-019-00619-1.

[178] N. M. Foumani, L. Miller, C. W. Tan, G. I. Webb, G. Forestier, and M. Salehi, "Deep Learning for Time Series Classification and Extrinsic Regression: A Current Survey," *ACM Comput. Surv.*, Feb. 2024, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0360-0300. DOI: 10.1145/3649448. [Online]. Available: https://doi.org/10.1145/3649448.

[179] 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.

[180] H. A. Dau et al., "The UCR Time Series Archive," *CoRR*, vol. abs/1810.07758, 2018, arXiv: 1810.07758. [Online]. Available: http://arxiv.org/abs/1810.07758.

[181] 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. DOI: 10.1109/IJCNN.2017.7966039.

[182] R. Killick, P. Fearnhead, and I. A. Eckley, "Optimal Detection of Changepoints With a Linear Computational Cost," *Journal of the American Statistical Association*, vol. 107, no. 500, pp. 1590–1598, 2012, Publisher: [American Statistical Association, Taylor & Francis, Ltd.], ISSN: 01621459. Accessed: Mar. 14, 2024. [Online]. Available: http://www.jstor.org/stable/23427357.

[183] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980, ISBN: 1432-0770. DOI: 10.1007/BF00344251. [Online]. Available: https://doi.org/10.1007/BF00344251.

[184] S. Ioffe and C. Szegedy, "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*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 448–456.

[185] K. Fukushima, "Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements," *IEEE Transactions on Systems Science and Cybernetics*, vol. 5, no. 4, pp. 322–333, 1969. DOI: 10.1109/TSSC.1969.300225.

[186] M. Lin, Q. Chen, and S. Yan, "Network In Network," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2014. [Online]. Available: http://arxiv.org/abs/1312.4400.

[187] Z. Cui, W. Chen, and Y. Chen, "Multi-Scale Convolutional Neural Networks for Time Series Classification," *CoRR*, vol. abs/1603.06995, 2016, arXiv: 1603.06995. [Online]. Available: http://arxiv.org/abs/1603.06995.

[188] 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. [Online]. Available: https://www.biodiversitylibrary.org/item/137258.

[189] 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. DOI: 10.1109/TSE.2022.3188005.

[190] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1495–1507, ISBN: 978-1-6654-5701-9. DOI: 10.1109/ICSE48619.2023.00130. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00130.

[191] T. C. Hesterberg, "What Teachers Should Know About the Bootstrap: Resampling in the Undergraduate Statistics Curriculum," *The American Statistician*, vol. 69, no. 4, pp. 371–386, 2015, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00031305.2015.1089789. DOI: 10.1080/00031305.2015.1089789. [Online]. Available: https://doi.org/10.1080/00031305.2015.1089789.

[192] D. P. Kingma and J. Ba, "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*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980.

[193] W. J. Youden, "Index for rating diagnostic tests," *Cancer*, vol. 3, no. 1, pp. 32–35, 1950. DOI: "https://doi.org/10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3".

[194] J. Chen, W. Shang, and E. Shihab, "PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1529–1544, 2022. DOI: 10.1109/TSE.2020.3023955.

[195] C. Laaber, T. Yue, and S. Ali, "Evaluating Search-Based Software Microbenchmark Prioritization," *IEEE Transactions on Software Engineering*, pp. 1–16, 2024. DOI: 10.1109/TSE.2024.3380836.

[196] M. Abdullah, L. Bulej, T. Bureš, V. Horký, and P. Tůma, "Early Stopping of Non-productive Performance Testing Experiments Using Measurement Mutations," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2023, pp. 86–93. DOI: 10.1109/SEAA60479.2023.00022.

[197] W. B. Langdon and M. Harman, "Optimizing Existing Software With Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, 2015. DOI: 10.1109/TEVC.2013.2281544.

[198] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2018. DOI: 10.1109/TEVC.2017.2693219.

[199] S. Wang, H. Hoffmann, and S. Lu, "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*, ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, 2022, pp. 459–471, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549136. [Online]. Available: https://doi.org/10.1145/3540250.3549136.

[200] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[201] I. Price et al., "Probabilistic weather forecasting with machine learning," *Nature*, vol. 637, no. 8044, pp. 84–90, Jan. 2025, ISBN: 1476-4687. DOI: 10.1038/s41586-024-08252-9. [Online]. Available: https://doi.org/10.1038/s41586-024-08252-9.

[202] M. U. G. Kraemer et al., "Artificial intelligence for modelling infectious disease epidemics," *Nature*, vol. 638, no. 8051, pp. 623–635, Feb. 2025, ISBN: 1476-4687. DOI: 10.1038/s41586-024-08564-w. [Online]. Available: https://doi.org/10.1038/s41586-024-08564-w.

[203] M. Tan, M. A. Merrill, V. Gupta, T. Althoff, and T. Hartvigsen, "Are Language Models Actually Useful for Time Series Forecasting?" In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://openreview.net/forum?id=DV15UbHCY1.

[204] O. Barkan, J. Benchimol, I. Caspi, E. Cohen, A. Hammer, and N. Koenigstein, "Forecasting CPI inflation components with Hierarchical Recurrent Neural Networks," *International Journal of Forecasting*, vol. 39, no. 3, pp. 1145–1162, 2023, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2022.04.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207022000607.

[205] D. Effrosynidis, E. Spiliotis, G. Sylaios, and A. Arampatzis, "Time series and regression methods for univariate environmental forecasting: An empirical evaluation," *Science of The Total Environment*, vol. 875, p. 162 580, 2023, ISSN: 0048-9697. DOI: https://doi.org/10.1016/j.scitotenv.2023.162580. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0048969723011968.

[206] F. Piccialli, F. Giampaolo, E. Prezioso, D. Camacho, and G. Acampora, "Artificial intelligence and healthcare: Forecasting of medical bookings through multi-source time-series fusion," *Information Fusion*, vol. 74, pp. 1–16, 2021, ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2021.03.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253521000592.

[207] A. Sagheer and M. Kotb, "Time series forecasting of petroleum production using deep LSTM recurrent networks," *Neurocomputing*, vol. 323, pp. 203–213, 2019, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2018.09.082. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218311639.

[208] H. Yan and H. Ouyang, "Financial Time Series Prediction Based on Deep Learning," *Wireless Personal Communications*, vol. 102, no. 2, pp. 683–700, Sep. 2018, ISSN: 1572-834X. DOI: 10.1007/s11277-017-5086-2. [Online]. Available: https://doi.org/10.1007/s11277-017-5086-2.

[209] J. H. Stock and M. W. Watson, "Forecasting Using Principal Components from a Large Number of Predictors," *Journal of the American Statistical Association*, vol. 97, no. 460, pp. 1167–1179, 2002, Publisher: [American Statistical Association, Taylor & Francis, Ltd.], ISSN: 01621459. Accessed: Oct. 25, 2023. [Online]. Available: http://www.jstor.org/stable/3085839.

[210] P. Chakraborty, M. Corici, and T. Magedanz, "System Failure Prediction within Software 5G Core Networks using Time Series Forecasting," in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2021, pp. 1–7. DOI: 10.1109/ICCWorkshops50388.2021.9473530.

[211] G. Box and G. M. Jenkins, *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.

[212] B. Billah, M. L. King, R. D. Snyder, and A. B. Koehler, "Exponential smoothing model selection for forecasting," *International Journal of Forecasting*, vol. 22, no. 2, pp. 239–247, 2006, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2005.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016920700500107X.

[213] S. J. Taylor and B. Letham, "Forecasting at Scale," *PeerJ Prepr.*, vol. 5, e3190, 2017. DOI: 10.7287/PEERJ.PREPRINTS.3190V1. [Online]. Available: https://doi.org/10.7287/peerj.preprints.3190v1.

[214] H. Hewamalage, C. Bergmeir, and K. Bandara, "Recurrent Neural Networks for Time Series Forecasting: Current status and future directions," *International Journal of Forecasting*, vol. 37, no. 1, pp. 388–427, 2021, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2020.06.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207020300996.

[215] R. W. Godahewa, C. Bergmeir, G. I. Webb, R. Hyndman, and P. Montero-Manso, "Monash Time Series Forecasting Archive," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. [Online]. Available: https://openreview.net/forum?id=wEc1mgAjU-.

[216] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "M5 accuracy competition: Results, findings, and conclusions," *International Journal of Forecasting*, vol. 38, no. 4, pp. 1346–1364, 2022, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2021.11.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207021001874.

[217] S. C. Yogi, V. K. Tripathi, and L. Behera, "Adaptive Integral Sliding Mode Control Using Fully Connected Recurrent Neural Network for Position and Attitude Control of Quadrotor," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 12, pp. 5595–5609, 2021. DOI: 10.1109/TNNLS.2021.3071020.

[218] E. C. Mengüç and N. Acır, "Kurtosis-Based CRTRL Algorithms for Fully Connected Recurrent Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 6123–6131, 2018. DOI: 10.1109/TNNLS.2018.2826442.

[219] K. Cho et al., "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. [Online]. Available: https://aclanthology.org/D14-1179.

[220] A. Heidari and D. Khovalyg, "Short-term energy use prediction of solar-assisted water heating system: Application case of combined attention-based LSTM and time-series decomposition," *Solar Energy*, vol. 207, pp. 626–639, Sep. 2020. DOI: 10.1016/j.solener.2020.07.008.

[221] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[222] A. Grattafiori et al., *The Llama 3 Herd of Models*, _eprint: 2407.21783, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783.

[223] S. Bubeck et al., *Sparks of Artificial General Intelligence: Early experiments with GPT-4*, _eprint: 2303.12712, 2023. [Online]. Available: https://arxiv.org/abs/2303.12712.

[224] A. Vaswani et al., "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, event-place: Long Beach, California, USA, Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 978-1-5108-6096-4.

[225] A. Garza, C. Challu, and M. Mergenthaler-Canseco, *TimeGPT-1*, _eprint: 2310.03589, 2024. [Online]. Available: https://arxiv.org/abs/2310.03589.

[226] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, Jan. 2020, Publisher: JMLR.org, ISSN: 1532-4435.

[227] E. Fuentetaja and D. Bagert, "Software Evolution from a Time-Series Perspective," in *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, ser. ICSM '02, USA: IEEE Computer Society, 2002, p. 226, ISBN: 0-7695-1819-2.

[228] W. Turski, "The reference model for smooth growth of software systems revisited," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 814–815, 2002. DOI: 10.1109/TSE.2002.1027802.

[229] M. Learn, *Blue screen data*, 01, Apr. 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/blue-screen-data.

[230] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation* (ITPro collection). A K Peters, 1998. [Online]. Available: https://books.google.it/books?id=EOrVzQEACAAJ.

[231] W. Wei, *Time Series Analysis: Univariate and Multivariate Methods, 2nd edition, 2006*. Jan. 2006, ISBN: 0-321-32216-9.

[232] A. Bauer, M. Züfle, S. Eismann, J. Grohmann, N. Herbst, and S. Kounev, "Libra: A Benchmark for Time Series Forecasting Methods," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21, event-place: Virtual Event, France, New York, NY, USA: Association for Computing Machinery, 2021, pp. 189–200, ISBN: 978-1-4503-8194-9. DOI: 10.1145/3427921.3450241. [Online]. Available: https://doi.org/10.1145/3427921.3450241.

[233] N. Kourentzes, "Intermittent demand forecasts with neural networks," *International Journal of Production Economics*, vol. 143, pp. 198–206, May 2013. DOI: 10.1016/j.ijpe.2013.01.009.

[234] M. Peixeiro, *Time Series Forecasting in Python*. Manning, 2022, ISBN: 978-1-61729-988-9.

[235] D. Dickey and W. Fuller, "Distribution of the Estimators for Autoregressive Time Series With a Unit Root," *JASA. Journal of the American Statistical Association*, vol. 74, Jun. 1979. DOI: 10.2307/2286348.

[236] H. Akaike, "A new look at the statistical model identification," *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, 1974. DOI: 10.1109/TAC.1974.1100705.

[237] S. Suhartono, "Time Series Forecasting by using Seasonal Autoregressive Integrated Moving Average: Subset, Multiplicative or Additive Model," *Journal of Mathematics and Statistics*, vol. 7, pp. 20–27, Jan. 2011. DOI: 10.3844/jmssp.2011.20.27.

[238] R. J. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*, English, 2nd. Australia: OTexts, 2018.

[239] N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj, and A. Iosifidis, "Deep Adaptive Input Normalization for Time Series Forecasting," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 9, pp. 3760–3765, 2020. DOI: 10.1109/TNNLS.2019.2944933.

[240] B. Cavallo, M. Di Penta, and G. Canfora, "An empirical comparison of methods to support QoS-aware service selection," in *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, event-place: Cape Town South Africa, New York, NY, USA: ACM, May 2010.

[241] A. Amin, L. Grunske, and A. Colman, "An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, event-place: Essen Germany, New York, NY, USA: ACM, Sep. 2012.

[242] F. Wilcoxon, "Individual Comparisons by Ranking Methods," in *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N. L. Johnson, Eds., New York, NY: Springer New York, 1992, pp. 196–202, ISBN: 978-1-4612-4380-9. DOI: 10.1007/978-1-4612-4380-9_16. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_16.

[243] K. O. McGraw and S. P. Wong, "A common language effect size statistic.," *Psychological bulletin*, vol. 111, no. 2, p. 361, 1992, Publisher: American Psychological Association.

[244] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "The M4 Competition: 100, 000 time series and 61 forecasting methods," *International Journal of Forecasting*, vol. 36, no. 1, pp. 54–74, Jan. 2020, Publisher: Elsevier BV. DOI: 10.1016/j.ijforecast.2019.04.014. [Online]. Available: https://doi.org/10.1016/j.ijforecast.2019.04.014.

[245] Y. Zhang and P. J. Thorburn, "Handling missing data in near real-time environmental monitoring: A system and a review of selected methods," *Future Gener. Comput. Syst.*, vol. 128, pp. 63–72, 2022. DOI: 10.1016/J.FUTURE.2021.09.033. [Online]. Available: https://doi.org/10.1016/j.future.2021.09.033.

[246] S. Kim, H. Kim, E. Yun, H. Lee, J. Lee, and J. Lee, "Probabilistic imputation for time-series classification with missing data," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 16 654–16 667. [Online]. Available: https://proceedings.mlr.press/v202/kim23m.html.

[247] J. S. Richman and J. R. Moorman, "Physiological time-series analysis using approximate entropy and sample entropy," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 278, no. 6, H2039–H2049, Jun. 2000, ISSN: 1522-1539. DOI: 10.1152/ajpheart.2000.278.6.h2039. [Online]. Available: http://dx.doi.org/10.1152/ajpheart.2000.278.6.h2039.

[248] F. Ullah, M. Bilal, and S. Yoon, "Intelligent time-series forecasting framework for non-linear dynamic workload and resource prediction in cloud," *Comput. Networks*, vol. 225, p. 109 653, 2023. DOI: 10.1016/J.COMNET.2023.109653. [Online]. Available: https://doi.org/10.1016/j.comnet.2023.109653.

[249] I. Svetunkov, *15.2 ARIMA order selection | Forecasting and Analytics with the Augmented Dynamic Adaptive Model (ADAM) - v2 — openforecast.org*, https://openforecast.org/adam/ARIMASelection.html, [Accessed 10-01-2026].

[250] *Selecting SARIMA Model Order Parameters — apxml.com*, https://apxml.com/courses/time-series-analysis-forecasting/chapter-5-seasonal-arima-sarima/selecting-sarima-order, [Accessed 10-01-2026].

[251] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," en, *Math. Program.*, vol. 45, no. 1-3, pp. 503–528, Aug. 1989, Publisher: Springer Science and Business Media LLC.

[252] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *Comput. J.*, vol. 7, no. 2, pp. 155–162, Feb. 1964, Publisher: Oxford University Press (OUP).

[253] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "STL: A Seasonal-Trend Decomposition Procedure Based on Loess (with Discussion)," *Journal of Official Statistics*, vol. 6, pp. 3–73, 1990.

[254] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997. DOI: 10.1109/78.650093.

[255] H. Wang, M. Li, and X. Yue, "Inclstm: Incremental ensemble lstm model towards time series data," *Computers and Electrical Engineering*, vol. 92, p. 107 156, Jun. 2021, ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2021.107156. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2021.107156.

[256] O. Anava, E. Hazan, S. Mannor, and O. Shamir, "Online learning for time series prediction," in *COLT 2013 - The 26th Annual Conference on Learning Theory, June 12-14, 2013, Princeton University, NJ, USA*, S. Shalev-Shwartz and I. Steinwart, Eds., ser. JMLR Workshop and Conference Proceedings, vol. 30, JMLR.org, 2013, pp. 172–184. [Online]. Available: http://proceedings.mlr.press/v30/Anava13.html.

[257] C. Willmott and K Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate Research*, vol. 30, 79–82, 2005, ISSN: 1616-1572. DOI: 10.3354/cr030079. [Online]. Available: http://dx.doi.org/10.3354/cr030079.

[258] M. Friedman, "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance," *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937, Publisher: ASA Website _eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1937.10503522. DOI: 10.1080/01621459.1937.10503522. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1937.10503522.

[259] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979, Publisher: [Board of the Foundation of the Scandinavian Journal of Statistics, Wiley], ISSN: 03036898, 14679469. Accessed: May 30, 2025. [Online]. Available: http://www.jstor.org/stable/4615733.

[260] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952, Publisher: [American Statistical Association, Taylor & Francis, Ltd.], ISSN: 01621459, 1537274X. Accessed: Mar. 25, 2025.

[261] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar. 1947, Publisher: Institute of Mathematical Statistics, ISSN: 0003-4851. DOI: 10.1214/aoms/1177730491.

[262] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. John Wiley & Sons, 2015.

[263] Z. Zamanzadeh Darban, G. I. Webb, S. Pan, C. Aggarwal, and M. Salehi, "Deep Learning for Time Series Anomaly Detection: A Survey," *ACM Comput. Surv.*, vol. 57, no. 1, Oct. 2024, Publisher: Association for Computing Machinery, ISSN: 0360-0300. DOI: 10.1145/3691338. [Online]. Available: https://doi.org/10.1145/3691338.

[264] J. G. D. Gooijer and R. J. Hyndman, "25 years of time series forecasting," *International Journal of Forecasting*, vol. 22, no. 3, pp. 443–473, 2006, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2006.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207006000021.

[265] H. Si et al., "TimeSeriesBench: An Industrial-Grade Benchmark for Time Series Anomaly Detection Models," in *IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2024, pp. 61–72. DOI: 10.1109/ISSRE62328.2024.00017. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSRE62328.2024.00017.

[266] S. Zhang et al., "Efficient KPI Anomaly Detection Through Transfer Learning for Large-Scale Web Services," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 8, pp. 2440–2455, 2022. DOI: 10.1109/JSAC.2022.3180785.

[267] A. D. Competition, *KPI Dataset*, 2018. [Online]. Available: https://github.com/NetManAIOps/KPI-Anomaly-Detection.

[268] *MALIAS Replication Package*, 2025. [Online]. Available: https://anonymous.4open.science/r/meta-multitask-F38C.

[269] S. Kong, M. Lu, J. Ai, and S. Wang, "Detection Software Content Failures using Dynamic Execution Information," *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 141–147, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238744449.

[270] Z. Zeng et al., "TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems," in *IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice*, 2023, pp. 258–269. DOI: 10.1109/ICSE-SEIP58684.2023.00029.

[271] F. Collopy and J. S. Armstrong, "Rule-Based Forecasting: Development and Validation of an Expert Systems Approach to Combining Time Series Extrapolations," *Manage. Sci.*, vol. 38, no. 10, pp. 1394–1414, Oct. 1992, Place: Linthicum, MD, USA Publisher: INFORMS, ISSN: 0025-1909.

[272] C. Lemke and B. Gabrys, "Meta-learning for time series forecasting and forecast combination," *Neurocomputing*, vol. 73, no. 10–12, pp. 2006–2016, Jun. 2010, Publisher: Elsevier BV, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2009.09.020. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2009.09.020.

[273] X. Wang, K. Smith-Miles, and R. Hyndman, "Rule induction for forecasting method selection: Meta-learning the characteristics of univariate time series," *Neurocomputing*, vol. 72, no. 10–12, pp. 2581–2594, Jun. 2009, Publisher: Elsevier BV, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2008.10.017. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2008.10.017.

[274] M. Abdallah, R. Rossi, K. Mahadik, S. Kim, H. Zhao, and S. Bagchi, "AutoForecast: Automatic Time-Series Forecasting Model Selection," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, ser. CIKM '22, 2022, pp. 5–14, ISBN: 978-1-4503-9236-5. DOI: 10.1145/3511808.3557241. [Online]. Available: https://doi.org/10.1145/3511808.3557241.

[275] M. Abdallah, R. A. Rossi, K. Mahadik, S. Kim, H. Zhao, and S. Bagchi, "Evaluation-free Time-series Forecasting Model Selection via Meta-learning," *ACM Trans. Knowl. Discov. Data*, vol. 19, no. 3, Mar. 2025, ISSN: 1556-4681. DOI: 10.1145/3715149. [Online]. Available: https://doi.org/10.1145/3715149.

[276] S. Wang, M. Rußwurm, M. Körner, and D. B. Lobell, "Meta-Learning For Few-Shot Time Series Classification," in *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, 2020, pp. 7041–7044. DOI: 10.1109/IGARSS39084.2020.9441016.

[277] A. Abanda, U. Mori, and J. A. Lozano, "Time series classifier recommendation by a meta-learning approach," *Pattern Recognition*, vol. 128, p. 108 671, Aug. 2022, Publisher: Elsevier BV, ISSN: 0031-3203. DOI: 10.1016/j.patcog.2022.108671. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2022.108671.

[278] J. Narwariya, P. Malhotra, L. Vig, G. Shroff, and T. V. Vishnu, "Meta-Learning for Few-Shot Time Series Classification," in *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*, ser. CoDS COMAD 2020, 2020, pp. 28–36, ISBN: 978-1-4503-7738-6. DOI: 10.1145/3371158.3371162. [Online]. Available: https://doi.org/10.1145/3371158.3371162.

[279] P. Pilyugina et al., "A Large-Scale Empirical Study of Aligned Time Series Forecasting," *IEEE Access*, vol. 12, pp. 131 100–131 121, 2024, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 2169-3536. DOI: 10.1109/access.2024.3458391. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2024.3458391.

[280] J. R. Rice, "The Algorithm Selection Problem," in *Advances in Computers Volume 15*, ISSN: 0065-2458, Elsevier, 1976, pp. 65–118. DOI: 10.1016/s0065-2458(08)60520-3. [Online]. Available: http://dx.doi.org/10.1016/S0065-2458(08)60520-3.

[281] K. Yi et al., "Frequency-domain MLPs are More Effective Learners in Time Series Forecasting," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 76 656–76 679. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/f1d16af76939f476b5f040fd1398c0a3-Paper-Conference.pdf.

[282] N. Shrestha, "Detecting Multicollinearity in Regression Analysis," *American Journal of Applied Mathematics and Statistics*, vol. 8, no. 2, pp. 39–42, Jun. 2020, Publisher: Science and Education Publishing Co., Ltd., ISSN: 2328-7306. DOI: 10.12691/ajams-8-2-1. [Online]. Available: http://dx.doi.org/10.12691/ajams-8-2-1.

[283] B. F. Darst, K. C. Malecki, and C. D. Engelman, "Using recursive feature elimination in random forest to account for correlated variables in high dimensional data," *BMC Genetics*, vol. 19, no. S1, Sep. 2018, Publisher: Springer Science and Business Media LLC, ISSN: 1471-2156. DOI: 10.1186/s12863-018-

0633-8. [Online]. Available: http://dx.doi.org/10.1186/s12863-018-0633-8.

[284] W. Liu, W. Zhang, Y. Jiang, S. Chang, and S. Zhang, "Learning Triple-View Representation Discrepancy for Multivariate Time Series Anomaly Detection with Multi-Scale Patching," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*, 2024, pp. 560–567. DOI: 10.1109/ICPADS63350.2024.00079.

[285] Y. Jing et al., "Diner: Interpretable Anomaly Detection for Seasonal Time Series in Web Services," *IEEE Transactions on Services Computing*, vol. 17, no. 5, pp. 2248–2260, 2024. DOI: 10.1109/TSC.2024.3422894.

[286] G. O. Ferreira, C. Ravazzi, F. Dabbene, G. C. Calafiore, and M. Fiore, "Forecasting Network Traffic: A Survey and Tutorial With Open-Source Comparative Evaluation," *IEEE Access*, vol. 11, pp. 6018–6044, 2023, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 2169-3536. DOI: 10.1109/access.2023.3236261. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2023.3236261.

[287] A. P. Jegannathan, R. Saha, and S. K. Addya, "A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform," in *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, IEEE, Jun. 2022, pp. 325–330. DOI: 10.1109/blackseacom54372.2022.9858271. [Online]. Available: http://dx.doi.org/10.1109/BlackSeaCom54372.2022.9858271.

[288] M. Zhang, X. Shi, J. Huang, L. Su, and Y. Zhang, "DynTrackr: A Robust Two-Stage Framework with Attribute Enhancement for KPI Anomaly Detection," in *IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2024, pp. 169–176. DOI: 10.1109/issrew63542.2024.00069. [Online]. Available: http://dx.doi.org/10.1109/ISSREW63542.2024.00069.

[289] Z. Yu et al., "AutoKAD: Empowering KPI Anomaly Detection with Label-Free Deployment," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, Oct. 2023, pp. 13–23. DOI: 10.1109/issre59848.2023.00063. [Online]. Available: http://dx.doi.org/10.1109/ISSRE59848.2023.00063.

[290] P. Malhotra, L. Vig, G. M. Shroff, and P. Agarwal, "Long Short Term Memory Networks for Anomaly Detection in Time Series," in *The European Symposium on Artificial Neural Networks*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:43680425.

[291] P. J. Rousseeuw and A. M. Leroy, *Robust Regression and Outlier Detection*, en. John Wiley & Sons, Dec. 1987.

[292] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. M. Shroff, "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection," *ArXiv*, vol. abs/1607.00148, 2016.

[293] Z. Wang et al., *Revisiting VAE for Unsupervised Time Series Anomaly Detection: A Frequency Perspective*, _eprint: 2402.02820, 2024. [Online]. Available: https://arxiv.org/abs/2402.02820.

[294] Z. Xu, A. Zeng, and Q. Xu, "FITS: Modeling Time Series with \10k\ Parameters," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=bWcnvZ3qMb.

[295] K. Boyd, K. H. Eng, and C. D. Page, "Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals," in *Advanced Information Systems Engineering*, ISSN: 1611-3349, Springer Berlin Heidelberg, 2013, pp. 451–466, ISBN: 978-3-642-38709-8. DOI: 10.1007/978-3-642-40994-3_29. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40994-3_29.

[296] L. Kuncheva, "A theoretical study on six classifier fusion strategies," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 2, pp. 281–286, 2002, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 0162-8828. DOI: 10.1109/34.982906. [Online]. Available: http://dx.doi.org/10.1109/34.982906.

[297] R. M. Cruz, R. Sabourin, and G. D. Cavalcanti, "META-DES.Oracle: Meta-learning and feature selection for dynamic ensemble selection," *Information Fusion*, vol. 38, pp. 84–103, Nov. 2017, Publisher: Elsevier BV, ISSN: 1566-2535. DOI: 10.1016/j.inffus.2017.02.010. [Online]. Available: http://dx.doi.org/10.1016/j.inffus.2017.02.010.

[298] P. R. G. Cordeiro, G. D. C. Cavalcanti, and R. M. O. Cruz, "Dynamic Ensemble Algorithm Post-Selection Using Hardness-Aware Oracle," *IEEE Access*, vol. 11, pp. 86 056–86 070, 2023, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 2169-3536. DOI: 10.1109/access.2023.3304912. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2023.3304912.

[299] R. Chandra, S. Goyal, and R. Gupta, "Evaluation of Deep Learning Models for Multi-Step Ahead Time Series Prediction," *IEEE Access*, vol. 9, pp. 83 105–83 123, 2021. DOI: 10.1109/ACCESS.2021.3085085.

[300] L Breiman, "Random Forests," *Machine Learning*, vol. 45, pp. 5–32, Oct. 2001. DOI: 10.1023/A:1010950718922.

[301] Y. G. Cinar, H. Mirisaee, P. Goswami, E. Gaussier, A. Aït-Bachir, and V. Strijov, "Position-Based Content Attention for Time Series Forecasting with Sequence-to-Sequence RNNs," in *Lecture Notes in Computer Science*, ISSN: 1611-3349, Springer International Publishing, 2017, pp. 533–544, ISBN: 978-3-319-70139-4. DOI: 10.1007/978-3-319-70139-4_54. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-70139-4_54.

[302] A. Kusiak, A. Verma, and X. Wei, "A data-mining approach to predict influent quality," *Environmental Monitoring and Assessment*, vol. 185, no. 3, pp. 2197–2210, Jun. 2012, Publisher: Springer Science and Business Media LLC, ISSN: 1573-2959. DOI: 10.1007/s10661-012-2701-2. [Online]. Available: http://dx.doi.org/10.1007/s10661-012-2701-2.

[303] K. Lin, Q. Lin, C. Zhou, and J. Yao, "Time Series Prediction Based on Linear Regression and SVR," in *Third International Conference on Natural Computation (ICNC 2007)*, vol. 1, 2007, pp. 688–691. DOI: 10.1109/ICNC.2007.780.

[304] W. Xu, H. Peng, X. Zeng, F. Zhou, X. Tian, and X. Peng, "A hybrid modelling method for time series forecasting based on a linear regression model and deep learning," *Applied Intelligence*, vol. 49, no. 8, pp. 3002–3015, Feb. 2019, Publisher: Springer Science and Business Media LLC, ISSN: 1573-7497. DOI: 10.1007/s10489-019-01426-3. [Online]. Available: http://dx.doi.org/10.1007/s10489-019-01426-3.

[305] S. B. Taieb, A. Sorjamaa, and G. Bontempi, "Multiple-output modeling for multi-step-ahead time series forecasting," *Neurocomputing*, vol. 73, no. 10, pp. 1950–1957, 2010, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2009.11.030. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231210001013.

[306] G. White, A. Palade, and S. Clarke, "Forecasting QoS Attributes Using LSTM Networks," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8. DOI: 10.1109/IJCNN.2018.8489052.

[307] D. Koutsandreas, E. Spiliotis, F. Petropoulos, and V. Assimakopoulos, "On the selection of forecasting accuracy measures," *Journal of the Operational Research Society*, vol. 73, no. 5, pp. 937–954, Apr. 2021, Publisher: Informa UK Limited, ISSN: 1476-9360. DOI: 10.1080/01605682.2021.1892464. [Online]. Available: http://dx.doi.org/10.1080/01605682.2021.1892464.

[308] E. H. M. Pena, M. V. O. de Assis, and M. L. Proença, "Anomaly Detection Using Forecasting Methods ARIMA and HWDS," in *2013 32nd International Conference of the Chilean Computer Science Society (SCCC)*, 2013, pp. 63–66. DOI: 10.1109/SCCC.2013.18.

[309] J. Qiu, Q. Du, W. Wang, K. Yin, and L. Chen, "Short-Term Performance Metrics Forecasting for Virtual Machine to Support Anomaly Detection Using Hybrid ARIMA-WNN Model," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2019, pp. 330–335. DOI: 10.1109/COMPSAC.2019.10228.

[310] M. S. Islam, W. Pourmajidi, L. Zhang, J. Steinbacher, T. Erwin, and A. Miranskyy, "Anomaly Detection in a Large-Scale Cloud Platform," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 150–159. DOI: 10.1109/ICSE-SEIP52600.2021.00024.

[311] X. Zhang et al., "Robust Log-Based Anomaly Detection on Unstable Log Data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, event-place: Tallinn, Estonia, New York, NY, USA: Association for Computing Machinery, 2019, pp. 807–817, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338931. [Online]. Available: https://doi.org/10.1145/3338906.3338931.

[312] H. Ren et al., "Time-Series Anomaly Detection Service at Microsoft," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19, event-place: Anchorage, AK, USA, New York, NY, USA: Association for Computing Machinery, 2019, pp. 3009–3017, ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330680. [Online]. Available: https://doi.org/10.1145/3292500.3330680.

[313] Q. Lin et al., "Predicting Node Failure in Cloud Service Systems," ser. ESEC/FSE 2018, event-place: Lake Buena Vista, FL, USA, New York, NY, USA: Association for Computing Machinery, 2018, pp. 480–490, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236060. [Online]. Available: https://doi.org/10.1145/3236024.3236060.

[314] D. Park, Y. Hoshi, and C. C. Kemp, "A Multimodal Anomaly Detector for Robot-Assisted Feeding Using an LSTM-Based Variational Autoencoder," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1544–1551, 2018. DOI: 10.1109/LRA.2018.2801475.

[315] N. Zhao et al., "Identifying Bad Software Changes via Multimodal Anomaly Detection for Online Service Systems," ser. ESEC/FSE 2021, event-place: Athens, Greece, New York, NY, USA: Association for Computing Machinery, 2021, pp. 527–539, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468543. [Online]. Available: https://doi.org/10.1145/3468264.3468543.

[316] G. Pang, K. M. Ting, and D. Albrecht, "LeSiNN: Detecting Anomalies by Identifying Least Similar Nearest Neighbours," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, 2015, pp. 623–630. DOI: 10.1109/ICDMW.2015.62.

[317] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying Impactful Service System Problems via Log Analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, event-place: Lake Buena Vista, FL, USA, New York, NY, USA: Association for Computing Machinery, 2018, pp. 60–70, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236083. [Online]. Available: https://doi.org/10.1145/3236024.3236083.

[318] Z. Chen et al., "Adaptive Performance Anomaly Detection for Online Service Systems via Pattern Sketching," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, event-place: Pittsburgh, Pennsylvania, New York, NY, USA: Association for Computing Machinery, 2022, pp. 61–72, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510085. [Online]. Available: https://doi.org/10.1145/3510003.3510085.

[319] E. Zhai, R. Piskac, R. Gu, X. Lao, and X. Wang, "An Auditing Language for Preventing Correlated Failures in the Cloud," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017, Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3133921. [Online]. Available: https://doi.org/10.1145/3133921.

[320] E. Zhai et al., "Check before You Change: Preventing Correlated Failures in Service Updates," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 575–589, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/zhai.

[321] S. Mehta et al., "Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 435–448, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/mehta.

[322] Y. Zhao, K. Damevski, and H. Chen, "A Systematic Survey of Just-in-Time Software Defect Prediction," *ACM Comput. Surv.*, vol. 55, no. 10, Feb. 2023, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0360-0300. DOI: 10.1145/3567550. [Online]. Available: https://doi.org/10.1145/3567550.

[323] J. Pool et al., "Lumos: A Library for Diagnosing Metric Regressions in Web-Scale Applications," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, event-place: Virtual Event, CA, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 2562–2570, ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3403306. [Online]. Available: https://doi.org/10.1145/3394486.3403306.

[324] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20, event-place: Seoul, South Korea, New York, NY, USA: Association for Computing Machinery, 2020, pp. 201–210, ISBN: 978-1-4503-7123-0. DOI: 10.1145/3377813.3381353. [Online]. Available: https://doi.org/10.1145/3377813.3381353.

[325] V. Cortellessa and L. Traini, "Detecting Latency Degradation Patterns in Service-Based Systems," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, event-place: Edmonton AB, Canada, New York, NY, USA: Association for Computing Machinery, 2020, pp. 161–172, ISBN: 978-1-4503-6991-6. DOI: 10.1145/3358960.3379126. [Online]. Available: https://doi.org/10.1145/3358960.3379126.

[326] D. Krushevskaja and M. Sandler, "Understanding Latency Variations of Black Box Services," in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW '13, event-place: Rio de Janeiro, Brazil, New York, NY, USA: Association for Computing Machinery, 2013, pp. 703–714, ISBN: 978-1-4503-2035-1. DOI: 10.1145/2488388.2488450. [Online]. Available: https://doi.org/10.1145/2488388.2488450.

[327] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance Debugging in the Large via Mining Millions of Stack Traces," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 145–155, ISBN: 978-1-4673-1067-3.

[328] T. B. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[329] J. Gong et al., "Language models for code optimization: Survey, challenges and future directions," *CoRR*, vol. abs/2501.01277, 2025. DOI: 10.48550/ARXIV.2501.01277. arXiv: 2501.01277. [Online]. Available: https://doi.org/10.48550/arXiv.2501.01277.

[330] X. He et al., "Swe-perf: Can language models optimize code performance on real-world repositories?" *CoRR*, vol. abs/2507.12415, 2025. DOI: 10.48550/ARXIV.2507.12415. arXiv: 2507.12415. [Online]. Available: https://doi.org/10.48550/arXiv.2507.12415.

[331] L. Yi, G. Gay, and P. Leitner, "An experimental study of real-life llm-proposed performance improvements," *CoRR*, vol. abs/2510.15494, 2025. DOI: 10.48550/ARXIV.2510.15494. arXiv: 2510.15494. [Online]. Available: https://doi.org/10.48550/arXiv.2510.15494.

[332] B. Šimek, D. Fiala, and M. Dostal, "Register-based and stack-based virtual machines: Which perform better in jit compilation scenarios?" *Software: Practice and Experience*, vol. 55, no. 11, 1896–1910, Aug. 2025, ISSN: 1097-024X. DOI: 10.1002/spe.70014. [Online]. Available: http://dx.doi.org/10.1002/spe.70014.

[333] *JVM Performance Comparison for JDK 21*, https://ionutbalosin.com/2024/02/jvm-performance-comparison-for-jdk-21/, [Accessed 05-12-2025].

[334] A. Ermshaus, P. Schäfer, and U. Leser, "Window size selection in unsupervised time series analytics: A review and benchmark," in *Advanced Analytics and Learning on Temporal Data: 7th ECML PKDD Workshop, AALTD 2022, Grenoble, France, September 19–23, 2022, Revised Selected Papers*, Grenoble, France: Springer-Verlag, 2022, 83–101, ISBN: 978-3-031-24377-6. DOI: 10.1007/978-3-031-24378-3_6. [Online]. Available: https://doi.org/10.1007/978-3-031-24378-3_6.

[335] P. Schäfer, "Scalable time series classification," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, 1273–1298, Nov. 2015, ISSN: 1573-756X. DOI: 10.1007/s10618-015-0441-y. [Online]. Available: http://dx.doi.org/10.1007/s10618-015-0441-y.

[336] A. Azlan, Y. Yusof, and M. F. M. Mohsin, "Determining the impact of window length on time series forecasting using deep learning," *International Journal of Advanced Computer Research*, vol. 9, no. 44, 260–267, Sep. 2019, ISSN: 2277-7970. DOI: 10.19101/ijacr.pid77. [Online]. Available: http://dx.doi.org/10.19101/IJACR.PID77.

[337] M. Loufakis et al., "Intelligent model management: Using reinforcement learning to detect data drift and retrain industrial machine learning systems," in *2024 26th International Conference on Digital Signal Processing and its Applications (DSPA)*, 2024, pp. 1–6. DOI: 10.1109/DSPA60853.2024.10510082.

[338] S. Dong, Q. Wang, S. Sahri, T. Palpanas, and D. Srivastava, "Efficiently mitigating the impact of data drift on machine learning pipelines," *Proc. VLDB Endow.*, vol. 17, no. 11, 3072–3081, Jul. 2024, ISSN: 2150-8097. DOI: 10.14778/3681954.3681984. [Online]. Available: https://doi.org/10.14778/3681954.3681984.

[339] X. Zhang et al., "Cross-dataset time series anomaly detection for cloud systems," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds., USENIX Association, 2019, pp. 1063–1076. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/zhang-xu.

[340] K. Bhukar et al., "Dynamic alert suppression policy for noise reduction in aiops," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*, ACM, 2024, pp. 178–188. DOI: 10.1145/3639477.3639752. [Online]. Available: https://doi.org/10.1145/3639477.3639752.

[341] Y. Tsubouchi and H. Tsuruta, "Metricsifter: Feature reduction of multivariate time series data for efficient fault localization in cloud applications," *IEEE Access*, vol. 12, pp. 37 398–37 417, 2024. DOI: 10.1109/ACCESS.2024.3374334.

[342] Y. Lyu, G. K. Rajbahadur, D. Lin, B. Chen, and Z. M. J. Jiang, "Towards a consistent interpretation of aiops models," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, Nov. 2021, ISSN: 1049-331X. DOI: 10.1145/3488269. [Online]. Available: https://doi.org/10.1145/3488269.

[343] V. Cortellessa, D. D. Pompeo, R. Eramo, and M. Tucci, "A model-driven approach for continuous performance engineering in microservice-based systems," *J. Syst. Softw.*, vol. 183, p. 111 084, 2022. DOI: 10.1016/J.JSS.2021.111084. [Online]. Available: https://doi.org/10.1016/j.jss.2021.111084.

[344] V. Cortellessa et al., "Introducing interactions in multi-objective optimization of software architectures," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, 181:1–181:39, 2025. DOI: 10.1145/3712185. [Online]. Available: https://doi.org/10.1145/3712185.

[345] G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana, "An exploratory study on architectural smell refactoring using large languages models," in *22nd IEEE International Conference on Software Architecture, ICSA - Companion, Odense, Denmark, March 31 - April 4, 2025*, IEEE, 2025, pp. 462–471. DOI: 10.1109/ICSA-C65153.2025.00070. [Online]. Available: https://doi.org/10.1109/ICSA-C65153.2025.00070.

Tesi redatta con il cofinanziamento dell'Unione europea-*NextGeneration EU*
Piano Nazionale di Ripresa e Resilienza Missione 4, Componente 2 –
Investimento 3.3. del PNRR – "Dottorati innovativi che rispondono ai fabbisogni di innovazione delle imprese" – CUP E11I22000200001