Performance Engineering in Agile/DevOps Development Processes

Ensuring Software Performance While Moving Fast

Luca Traini

Department of Information Engineering,
Computer Science and Mathematics
University of L'Aquila

This dissertation is submitted for the degree of
Doctor of Philosophy

Advisor:
Vittorio Cortellessa

Co-Advisor:
Henry Muccini

May 2021

Abstract

Agile principles and DevOps practices play a pivotal role in modern software development. These methodologies aim to improve software organization productivity while preserving the quality of the produced software. Unfortunately, the assessment of important non-functional software properties, such as performance, can be challenging in these contexts. Frequent code changes and software releases make impractical the use of classical performance assurance approaches. Moreover, many performance issues require highly specific conditions to be detected, which may be difficult to replicate in a testing environment.

This thesis investigates and tackles problems related to performance assessment of software systems in the context of Agile/DevOps development processes. Specifically, it focuses on three aspects.

The first aspect concerns practical and management problems in handling performance requirements. These problems were investigated through a 6-months industry collaboration with a large software organization that adopts an Agile software development process. The research was conducted in line with ethnographic research, which guided towards building knowledge from participatory observations, unstructured interviews and reviews of documentations. The study identified a set of management and practical challenges that arise from the adoption of Agile methodologies.

The second aspect concerns the impact of refactoring activities on software performance. Refactoring is a fundamental activity in modern software development, and it is a core development phase of many Agile methodologies (e.g., Test-Driven Development and Extreme Programming). Nevertheless, there is little knowledge about the impact of refactoring operations on software performance. This thesis aims to fill this gap by presenting the largest study to date that investigates the impact of refactoring on software performance, in terms of execution time. The change history of 20 Java open-source systems was mined with the goal of identifying commits in which developers have implemented refactoring operations impacting code components that are exercised by performance benchmarks. Through a quantitative and qualitative analysis, the impact of (different types of) refactoring on execution times were unveiled.

The results showed that the impact of refactoring on execution time varies depending on the refactoring type, with none of them being 100% "safe" in ensuring that there is no performance regression. Some refactoring types can result in substantial performance regression and, as such, should be carefully considered when refactoring performance-critical parts of a system.

The third aspect concerns the introduction of techniques for performance assessment in the context of DevOps processes. Due to the fast-faced release cycle and the inherently non-deterministic nature of software performance, it is often unfeasible to proactively detect performance issues. For these reasons, today, the diagnosis of performance issues in production is a fundamental activity for maintaining high-quality software systems. This activity can be time-consuming, since it may require thorough inspection of large volumes of traces and performance indices. This thesis introduces two novel techniques for automated diagnosis of performance issues in service-based systems, which can be easily integrated into DevOps processes. These techniques are evaluated, in terms of effectiveness and efficiency, on a large number of datasets generated for two case study systems, and they are compared to two state-of-the-art techniques and three general-purpose clustering algorithms. The results showed that baselines were outperformed with a better and more stable effectiveness. Moreover, the presented techniques showed to be more efficient on large datasets when compared to the most effective baseline.

Agile/DevOps development processes pose significant technical and management challenges for performance engineering. Although frequent code changes and continuous refactoring are likely sources of performance regressions, a proactive and exhaustive assessment of software performance is unrealistic in these contexts. Hence, the ability to quickly detect performance regressions in production becomes critical. The techniques proposed in this thesis take a step forward in this direction by allowing, through automated execution, early detection of performance issues, and thereby mitigating the potential adverse effects of fast-paced release cycles on software performance.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In the last decades, the principles behind the Agile Manifesto [12] have profoundly changed the way software is produced. Software development methodologies inspired by these principles (*e.g.,* Scrum and Kanban) are today widely adopted[1], and are gradually replacing the traditional waterfall approach. Indeed, Agile Software Development (ASD) better meets the dynamic nature of today's software development business and the significant pressure to deliver fast to the market [107]. DevOps has also emerged in recent years to complement the ASD effort in ensuring better software productivity[2], thus providing a set of principles and practices to enable faster software release cycles [71].

However, although ASD and DevOps have successfully enabled companies to address the current fast-to-market trend, their cost in terms of software quality is still disputable [107]. In the past years, researchers criticized ASD for mainly focusing on functional aspects and neglecting non-functional quality attributes [102, 61], and highlighted a lack of well-defined practices to effectively manage non-functional quality assurance [2, 3, 13, 14, 69].

Short iteration cycles and time constraints minimize the focus on addressing non-functional requirements [13, 3], and hamper the adoption of traditional quality assurance approaches. Software performance assurance, for example, can be especially challenging in these contexts [102, 137]. Common performance assessment techniques (*e.g.,* load testing [62]) are often too time-consuming for fast-paced release cycles. Also, traditional agile mentality suggests to consider software performance late in the development [8, 50], while, on the contrary, classical performance engineering literature

[1] 15^{th} State of Agile Survey. https://bit.ly/3azEj5r

[2] 2019 Accelerate State of DevOps Report. https://bit.ly/2ZuPfLf

warns that the late identification of performance issues may induce major performance failures and expensive reworks [119].

This thesis tackles three different aspects related to software performance assessment in the context of ASD/DevOps processes.

The first aspect concerns practical and management problems in handling performance assurance in ASD. Although several studies have broadly investigated non-functional quality assurance in these contexts, there is still little specific knowledge on the practices and challenges of performance assurance in ASD. Moreover, prior work present results based on data collected through interviews and surveys, but it does not examine how agile teams tackle non-functional quality assurance in their daily work. Motivated by the above issues, this thesis presents the first empirical study investigating performance assurance practices and challenges in ASD daily work. The research was conducted by means of a 6-months ethnographic study [116] in a Research & Development division of a large company, which used the Scrum framework [107] to support their agile practices. During the first three months, we gained understanding on the performance assurance practices and problems through observation of daily work and meeting sessions, individual interviews, participation in demo sessions, process workshops and review of documentation. In the last three months, the researcher actively participated to the improvement of the performance assurance process. Ethnography focus on daily activities helped to capture a holistic view of the performance assurance process, and to distill promising research directions related to three broad challenges: *managing performance assessment activities*, *continuous performance assessment* and *determining the performance assessment effort*.

The second aspect concerns the impact of software refactoring on performance. Refactoring is a core activity of modern software development, and it is a foundational part of many Agile development methodologies (*e.g.,* Extreme Programming and Test-Driven Development). Refactoring aims at improving the maintainability of source code without modifying its external behavior [49]. Previous works [131, 90, 11] proposed approaches to recommend refactoring solutions to software developers. The generation of the recommended solutions is guided by metrics acting as proxy for maintainability (e.g., number of code smells removed by the recommended solution). However, these approaches ignore the impact of the recommended refactorings on other non-functional requirements, such as performance. Indeed, little is known about the impact of refactoring operations on software performance. A second goal of this thesis is to fill this gap by presenting the largest study to date to investigate the impact of refactoring on software performance, in terms of execution time. The change history of

20 systems that defined performance benchmarks in their repositories was mined, with the goal of identifying commits in which developers implemented refactoring operations that impact code components that are exercised by the performance benchmarks. Through a quantitative and qualitative analysis, the thesis unveils the impact of (different types of) refactoring on execution times. Results showed that refactoring can have a substantial impact on execution time, both in a positive and negative way. Moreover, certain types of refactorings are more prone to degrade execution time and should be carefully performed in performance-critical systems.

The third aspect concerns the introduction of techniques for performance assessment in the context of DevOps processes. In order to support a fast-paced release cycle, IT organizations often employ several independent teams that are responsible "*from development to deploy*" [99] of loosely coupled independently deployable services. Service owners are usually responsible and accountable for meeting Service Level Objectives (SLOs) on Key Performance Indicators (KPIs). Given the frequency of software releases and the complexity of these systems, it's often unfeasible to proactively detect performance issues in a testing environment [134]. Software engineers and performance analysts continuously monitor KPIs and execution traces on run-time systems to identify symptoms of potentially relevant performance issues that lead to SLOs violations. The identification of such symptoms is often critical: a request may involve several Remote Procedure Calls (RPC) and the number of performance traces and metrics to analyze can be huge.

This thesis aims to tackle this challenge by presenting *LagranDe* and *DeLag*, two novel automated techniques for detecting *Latency Degradation Patterns*, *i.e.,* RPC execution time behaviors that are correlated with SLOs violation. The effectiveness and efficiency of the presented techniques are compared to those of state-of-the-art approaches and general purpose clustering algorithms. A first preliminary evaluation showed that LagranDe outperforms in terms of effectiveness both a state-of-the-art technique and three clustering algorithms, especially when RPC execution times are not very regular and vary over time. A more comprehensive evaluation demonstrated that DeLag provides (very often) better and (always) more stable effectiveness than LagranDe, two state-of-the-art approaches and two clustering algorithms. Both LagranDe and DeLag can be integrated in a DevOps process to enable continuous performance assessment of service-based systems.

The thesis is structured as follows:

Chapter 2 presents the ethnographic study on performance assurance practices and challenges in ASD. It describes the performance assurance practices of the case study organization, and reports the key challenges and promising research directions identified during the study.

Chapter 3 presents the study on the impact of refactoring on software performance, which shows and discusses the effects, in terms of execution time, of (different types of) refactoring operations in the change history of 20 open source Java projects.

Chapter 4 provides background on automated performance issue diagnosis and presents the concept of *Latency Degradation Pattern*, which is used throughout the remainder of the thesis.

Chapter 5 presents LagranDe, a first automated technique that uses F1-score optimization to detect *Latency Degradation Patterns.* The chapter outlines the components of LagranDe and presents a preliminary experimental evaluation.

Chapter 6 presents DeLag, a second automated technique for detecting *Latency Degradation Patterns* based on multi-objective optimization. The chapter outlines DeLag workflow and components along with a comprehensive experimental evaluation.

Chapter 7 presents final remarks and concludes this thesis.

Chapter 2

Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study

This chapter presents an empirical study on performance assurance practices and challenges in Agile Software Development. The article based on this chapter is currently under review [126].

2.1 Introduction

Agile Software Development (ASD) advocate early and continuous delivery of working software [12]. One challenge of this "*move fast*" mentality is to ensure software quality [106]. In the past years, Agile methodologies have been criticized for mainly focusing on functional aspects and neglecting non-functional quality attributes (*e.g.,* security, performance, usability) [61, 3]. For this reason, studies has been conducted to investigate challenges in non-functional quality assurance in the ASD context [2, 3, 13, 14]. However, while these studies investigate non-functional attributes in general, there is still little knowledge on practices and challenges to assess specific non-functional quality attributes, such as performance.

Poor software performance impacts user engagement and satisfaction [19], wastes computational resources, and lowers system throughput. Since the early days of ASD, researchers expressed concerns about how to inject performance assurance activities in Agile iterations [137]. Indeed, due to their time-consuming nature, common performance assurance practices (e.g., load testing [62]) are often unsuitable for ASD. Moreover, mentality and principles behind Agile often contradict common performance engineering conjectures. For example, Fowler and Beck, namely two signatories of the Agile Manifesto [12], suggest to consider software performance only after making the software clear and maintainable [8, 50], thereby conceptually reflecting the famous quote from Knuth that "*Premature optimization is the root of all evil in programming*" [73]. On the other hand, classical performance engineering literature argues that the "*fix-it-late*" attitude is one of the causes of major performance failures [119]. To best of our knowledge, the only attempt to tackle this problem was made by Ho et al. [27], which proposed an evolutionary model for performance requirements specifications and corresponding validation testing that can be integrated into ASD.

Yet, after two decades of ASD, there is still a lack of empirical knowledge on performance assurance in this context. In order to contribute to fill this gap, we present in this chapter the first empirical study investigating performance assessment practice and challenges in ASD. We carried out an ethnographic study to examine practices, problems and challenges of performance assurance in a large company employing ASD for 6 months. We collected data through observation sessions of daily work and meetings, individual interviews and participation in demo sessions and process workshops. Thereafter, we have qualitatively analyzed these data to extract performance assurance practices and challenges.

2.2 Study context and design

This study was carried out in a Research & Development (R&D) division of *ECorp*[1], based in Italy, where more than 120 people work on building *MESPlatform* [1], *i.e.,* a software platform for the manufacturing industry domain. *MESPlatform* provides foundation for building Manufacturing Execution Systems [89](MES), which allow industries to digitalize their manufacturing chains by providing a real-time software layer to track and document the transformation of raw materials to finished goods.

The software development division was composed by 13 agile teams. The teams used the Scrum framework to support their agile practices [107]. Each team had between 8 and 10 developers, one of whom played the additional role of Scrum Master. Three additional teams were responsible of non-functional software quality aspects: a

[1] Due to the sensitivity of the results presented here-in, the organization chose to stay incognito. Therefore, in this thesis, we use fictitious names of the company and the product.

UX team, a security team and an enterprise testing team. The latter was in charge of performing highly complex tests to assess important non-functional quality attributes, such as reliability, robustness and performance. One head of development, three product owners, one product manager, three software architects, one release manager, one quality assurance specialist and two Agile coaches were other professionals involved in the software development process. Seven agile teams were distributed in other countries, i.e. India and Romania, while others agile teams and professionals were co-located in the same building in Italy. Development followed 3-week sprints, or iterations, and the teams used Scrum ceremonies, *e.g.,* sprint planning meetings, stand-up meetings, demo sessions and retrospectives. Software release happened every 6 sprints (roughly every 4 months).

In order to formalize the study, we used the five ethnographic dimensions as proposed by Sharp et al. [116]:

1. observation type (participant or not) was mixed. In the first phase of the study non-participant observation was used, with the researcher asking questions and observing individuals performing their tasks; In the second phase of the study, the researcher actively participated to meetings with the goal of improving the process;

2. Duration of field study was 6 months;

3. Space and location where the observation happened was the R&D labs of *ECorp* where *MESPlatform* was developed (Italy);

4. No specific theoretical underpinnings were used;

5. The ethnographer intent was to understand the current process for performance assurance and the challenges in improving this process;

The study started with four individual meetings to gain an overview of the software development process, the product and the software architecture. These meetings involved the head of software development, a product owner, a scrum master and a software architect.

After these preliminary meetings, the investigation on performance assurance practices started. The first unstructured interview was conducted with the head of development to gain an overview of the current process for performance assurance and the parts involved in this process (e.g., teams and professionals). After this interview, the researcher agreed with the head of development to spend two months of daily work observations within an agile team in charge of performance testing activities. The researcher observed the team in their daily work activities, such as scrum ceremonies (e.g, stand-up meeting, sprint plan and retrospectives) and technical activities (e.g., performance testing and debugging). Other two unstructured interview sessions were carried out with the quality assurance specialist to gain a detailed picture of the performance assurance process. Additionally, the researcher participated to all the meetings that were related to performance assurance aspects. These meetings involved agile teams, product owners, software architects, quality assurance specialists and the enterprise testing teams. The data was collected through different media like handwritten notes, photographs and digital copies of documents and artifacts.

After the first three months, the head of development proposed to start an investigation to identify opportunities for improvement in the performance assurance process. To this aim, a series of meetings involving product owners, software architects, quality assurance specialist, head of development, a scrum master, an external consultant and the enterprise testing team were held. The researcher actively participated to these meetings, thereby influencing the outcome of the research.

After these meetings, two activities were identified to improve the performance assurance process. The researcher actively participated to both these activities, which mainly involved meetings with product owners and software architects.

In order to complement data gathered from interviews, meetings and observations, the researcher also analyzed process and software documentation in Confluence[2] and Azure DevOps Server[3](ADS), which supported the R&D division in holding all the software development information (e.g., product backlog, test plans and wikis).

The collected data were then analyzed to derive (i) the practices adopted by the organization for performance assessment along with their problems, and (ii) the challenges in improving the performance assessment process.

2.3 State-of-practice

Overall performance assessment activities still followed a waterfall-like process. In this section, we first describe the performance assessment process, and then we report problems that arose from the use of such process.

[2] Atlassian Confluence https://www.atlassian.com/software/confluence

[3] Microsoft Azure DevOps Server, https://azure.microsoft.com/it-it/services/devops/server/

2.3.1 Performance Assessments Process

In the following, we outline the main components of the performance assessment process. Specifically we describe (i) the Performance Requirements (PR) *MESPlatform* was subject to, (ii) the process used to manage performance assessment activities, (iii) the documentation approach and (iv) the PR verification process.

Performance Requirements

MESPlatform was subject to 92 PRs, where each PR involved a set of testing scenarios that were used to evaluate the performance of a specific system operation. Each testing scenario was usually associated to a target (e.g., minimal expected throughput, maximal expected response time), which was defined according to customer agreements, or based on the system knowledge of product owners and software architects. In some PRs no targets were specified. These PRs were called benchmarks and were used to assess performance of new functionalities or to compare the performance of particular system operations across different software versions. Each PR described the system operation under test and the configurations required to each testing scenario, such as virtual machine specifications (e.g., number of CPU cores, memory size), database size (i.e, records stored in the database) and number of concurrent users. Overall there were more than 300 testing scenarios across the 92 PRs. Some of these tests required just few seconds of execution while others required more than one day. PRs involved four different types of performance tests: UI tests, web tests, load tests and mixed tests. Performance UI tests were used to measure the time needed to execute a particular task in the *MESPlatform* UI by simulating real user interaction through Selenium [4]. The advantage of these tests was their representativeness of the true usage of the application, as the web application is actually executed in a browser, thereby considering also UI rendering time in the response time measurement. Unfortunately, these tests are usually demanding in terms of maintenance, as even a tiny UI change may corrupt them. According to a senior software tester, it was rare that UI tests were used without additional maintenance tasks from one release to another. Performance web tests, instead, involved a series of HTTP requests that simulated a single user interaction. In this type of test, only server side response time (or throughput) was measured, while the UI rendering time was ignored. Load tests [62] were used to simulate many users that interact with the system at the same time. Load tests involved the simultaneous execution of multiple web tests that simulate multiple users

[4] Selenium WebDriver, https://www.selenium.dev

making multiple simultaneous HTTP requests. Mixed performance tests, instead, were UI tests combined with load tests, which enabled response time measurements of UI user interactions on the system under load.

PRs management

New PRs were usually defined by individual product owners or software architects based on their system knowledge or according to customer needs. On the other hand, the deprecation of a PR usually required acknowledgements among multiple architects and product owners. Overall, performance assurance activities still followed a waterfall-like process, i.e. performance tests were executed before each software release. At the best case, every PR was tested once before each release, however, due to time and resources constraints, the exhaustively assessment of PRs before release was usually impractical. As a matter of fact, at the time of study, no more than 35% of PRs were addressed per release (see Fig. 2.1).

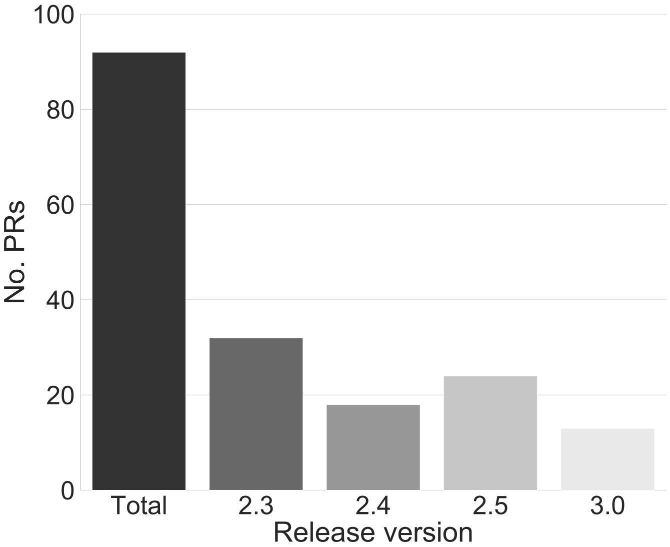


Fig. 2.1 PRs tested per releases. The first bar represents the total number of PRs, while the others represent PRs tested in a particular release version.

In order to ensure the assessment of relevant PRs before releasing, the company employed a priority mechanism, where each PR was prioritized with a number ranging from 1 to 4, with 1 being the highest priority and 4 the lowest. PR priorities were usually not stable across releases, since the development activities performed within a release cycle may often change the relevance of some PRs. According to the quality assurance specialist, priorities were usually updated before each release in a long one-day meeting involving software architects and product owners.

Performance tests were usually performed by the enterprise testing team, which chosen performance tests to execute according to priorities and time/resource constraints. In some cases, product owners specifically asked mandatory assessment of certain PRs. Some PRs were also assigned to a "special" agile team, called *ChogoRi*, which performed complementary performance assurance activities along with feature development and other operational tasks.

Documentation

PRs were documented through a custom artifact of ADS[5], which was specifically created to document Non-Functional Requirements (NFR). The catalog of PRs was accessible through ADS, and each PRs artifact contained an ID, a title, a description and the software versions in which the NFR was tested. The description contained the performance testing type (e.g., UI, load test), the description of system operation(s) under test, and a set of testing scenarios along with their detailed descriptions (e.g., machine specs, number of concurrent users). Supplementary information on PR was held in Confluence wikis. The enterprise testing team used test plans in ADS to manage and document the execution of performance tests, while the *ChogoRi* team adopted user stories[6]. Both test plans and user stories contained a reference to the PR ID.

PR verification

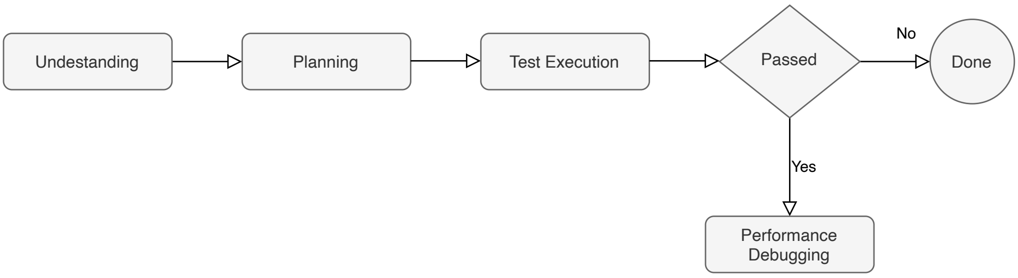


Fig. 2.2 PR verification process.

[5] Test plans, Azure DevOps Service. https://bit.ly/3aRlmdv
[6] User story, Agile Alliance. https://bit.ly/369HxtY

Fig. 2.2 outlines the main phases of PR verification, as they were observed in the *ChogoRi* team daily work. In the first phase, the team analyzed the PR description to gain understanding on the required testing scenarios. Often, in case of ambiguous PR description, architects or software developers helped the team in this process. In the planning phase, the work needed for PR verification was divided into tasks, such as environment configuration or test development, and assigned to team members. In the third phase, performance tests were executed while collecting application logs and performance indices (e.g., throughput, response time, CPU and memory utilizations). The operational data was then analyzed by the team to identify relevant anomalies and to verify PRs targets. In the case of benchmark PR (*i.e.,* no target specified), performance test results were typically compared to a baseline, *e.g.,* results of the latest release tested. If results did not met targets or showed relevant performance deviation compared to baseline, they were reported to architects and/or product owners, which, after a careful evaluation, could confirm the performance bug. Similarly to functional bugs, performance bugs were tracked in ADS along with a level of severity[7]. The severity defines whether the performance bug had to be immediately resolved or it could be addressed in later development stages (according to priorities).

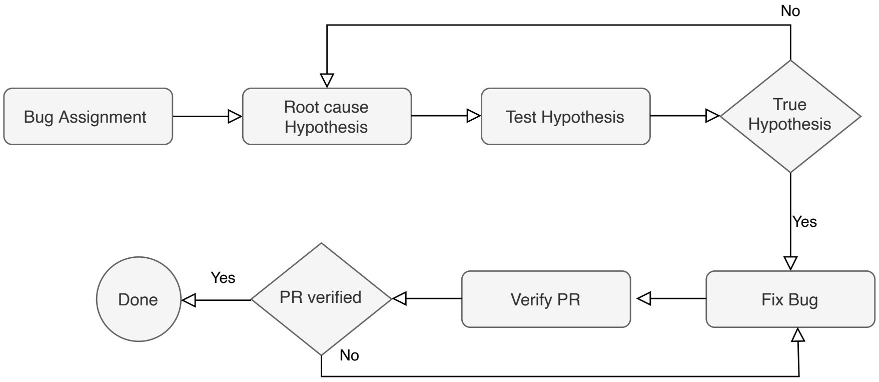


Fig. 2.3 Performance debugging process.

Fig. 2.3 outlines the performance debugging process as observed in the *ChogoRi* team. In order to assign the bug to the proper development team, the system components that are affected by the bug had to be identified. To this end, the *ChogoRi* team analyzed performance indices and/or log statements, that are somehow related to the performance issue, to pinpoint the software components that are affected by the performance bug. Often, the knowledge on the system by the *ChogoRi* team might

[7] Azure DevOps Service, bug management. https://bit.ly/3thSZgZ

not be sufficient to perform this task. Hence, in these cases, they were helped by software architects and developers. Once the affected components were identified, the bug was assigned to the proper development team, which thoroughly study the bug symptoms to identify the likely root cause. The root causing process usually involved three steps: 1) making a root cause hypothesis, 2) changing the source code according to the hypothesis and 3) validating the hypothesis by re-running performance tests on the modified system snapshot. Often this process had to be repeated several times in order to identify the real root cause. Once the root cause has been determined, the development team actually resolves the bug, and subsequently the performance testing team re-reruns testing scenarios to verify the PR. If the PR was met, then the bug could be marked as "resolved".

2.3.2 Problems

In the following, we report the main problems that were observed within the performance assurance process. We categorize these problems in two broad categories: PR management and lack of early feedback.

PR management

After a thorough analysis of test plans, user stories and PRs, we noticed that PR management was not working as expected. Although the number of PRs continued to grow from one release to another, due to the new features added to *MESPlatform*, most of them were often not verified. The cause of this phenomenon was that, while adding a new PR was a relatively cheap process for the organization (since it required the decision of single product owner or architect), the deprecation of a PR required the agreement of multiple product owners and architects. As a consequence, product owners kept adding PRs but their deprecation rarely happened, and, from time to time, the PRs list became unmanageable. Moreover, the process overhead introduced by this high number of PRs was often not justified by their usefulness. For example, we found that 35 PRs were never tested in the last 6 releases (see Fig. 2.4), i.e., since ~2 years.

The increased number of PRs also impacted the prioritization mechanism which was showing its vulnerability. Indeed, the analysis of the test plans and user stories reported a lack of alignment between priorities and what was actually tested (see Fig. 2.5). For example, several PRs with priority 1 (i.e., the highest) were not tested in the last release, while others with priority 4 (i.e., the lowest) were. When the researcher asked the quality assurance specialist why this happened, she replied that meetings for

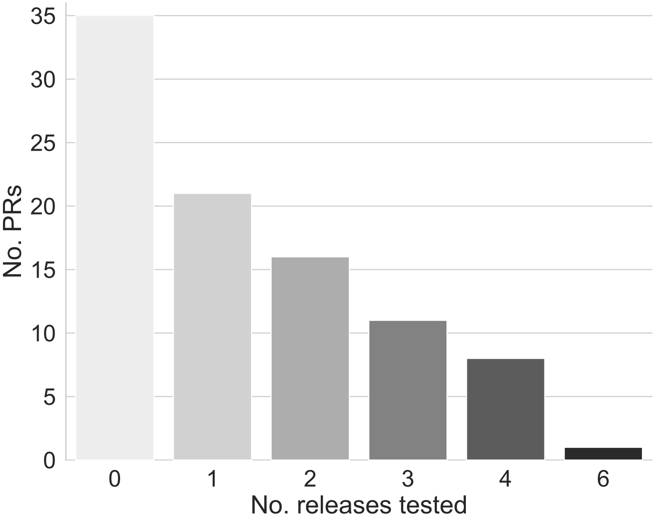


Fig. 2.4 Number of times PRs have been tested in the last 6 releases. The x label represents the amount of releases tested, while the y label represents the number of PRs that have been tested x times in the last 6 releases.

updating priorities were not held in the last two releases, therefore priorities might be obsolete. Indeed, product owners and software architects were usually overloaded with many relevant tasks in the organization, hence keeping them busy for a whole work day to update NFRs priorities was unfeasible in the last releases.

Nevertheless, according to the quality assurance specialist, at that time, the criteria used to select PRs for a particular release was not clear. We asked clarification to the enterprise team and product owners, and it turned out that several PRs were autonomously chosen by the enterprise testing team, apparently, according to their practical convenience. The lack of well established practices to manage PRs led to a suboptimal selection of performance tests, which compromised the performance assessment of *MESPlatform*. As a matter of fact, at the time of the study, several informants in the organization reported that other R&D divisions in *ECorp* that used *MESPlatform* to build other products, were experiencing severe performance issues. Moreover, while analyzing the product backlog, we found that an entire epic was devoted to improving *MESPlatform* performance, hence suggesting that a massive rework was planned. Note that epic is the larger unit of work in Scrum, and it is used as a placeholder that represents work that cannot be finished within a sprint (i.e., 3 weeks).

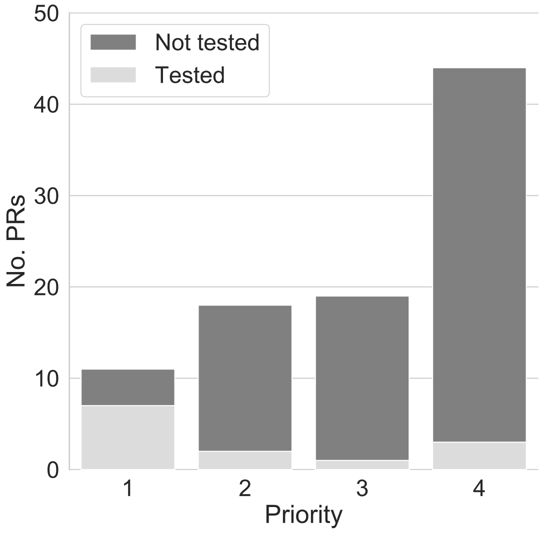


Fig. 2.5 Number of PRs tested (and not tested) in the latest release, grouped by priority.

Lack of early feedback

While functional tests were continuously executed, both in the continuous integration pipeline and in nightly builds, performance tests were executed at most once per release. Nevertheless, from one release to another, thousands of code changes were performed that might potentially affect software performance. This lack of performance feedback during development might hide potentially harmful performance issues, which could eventually lead to expensive maintenance activities. Moreover, a higher number of code changes usually implies a higher number of potential causes of performance regression, which makes harder analysis and problem resolution. In order to provide a better understanding of this problem, in the following we report the challenges observed by the researcher, in the *ChogoRi* team, during a complex PR verification.

The PR was derived from a real world scenario of an *MESPlatform* customer, and it involved a set of load testing scenarios. For each scenario, a target was specified to define the minimal expected throughput for a specific system operation. After setting up the environment (i.e., the virtual machine, the *MESPlatform* instance and its databases) and launching performance tests, the tester found that PR targets were not met. On the other hand, in the previous release, the same PR was verified without reporting issues, which implied that performance regression was introduced during the development of last release. The analysis of performance indices and logs found that the response time of the analyzed operation kept increasing during the test, and, in the meantime, the CPU utilization temporarily drop to zero. The symptoms of the performance issue were then reported to software architects and product owners, which, after a careful analysis, classified the problem as a critical bug that should have been immediately resolved. The bug was assigned to a development team who had worked, during the last release development, on the components apparently affected by the performance issue. Unfortunately, these software components had been subject to several changes since the previous release, thereby implying a potentially large set of potential root causes. As a matter of fact, the truly identification of the root cause required several attempts and about a week of work, in which the development team performed experimental code changes to the system snapshot and the performance testing team re-launched tests against the system snapshot to evaluate its performance. An earlier performance feedback would have probably implied less iterations to isolate the problem, due to a lower number of potential code changes, thereby enabling a faster root cause detection.

The late detection of the bug also impacted the effort and the quality of debugging. The performance bug was caused by a well known problem in .NET [8] asynchronous programming called "threadpool starvation" [9]. The root cause was the introduction of a synchronous call in the critical path of a system operation. Unfortunately, after the introduction of the bug, several changes were performed on the system that made difficult to fix the synchronous call. The impact of the bug was partially hampered through a work-around[9], which allowed the PR tests to pass. However, due to time-pressure and rework complexity, the actual resolution of the bug was unfeasible. An earlier identification of the problem would have probably enabled software developers to be more aware of the potential impact of their design decision on software performance, thereby avoiding potential technical debt.

2.4 Improving performance assessment

During the last three months of this empirical study, a series of workshops and meetings were held with the goal of improving the performance assessment process. Several proposals were discussed to tackle the problems faced by the organization, some of them are briefly discussed in the following.

[8] Microsoft .NET. https://bit.ly/2Muc3If

[9] NET Core, ThreadPool Starvation. http://bit.ly/3ozYnII

One proposal was to assign performance assessment activities directly to development teams to enable early assessment of development activities as soon as they are completed. The proposal was discarded due to various reasons. For instance, according to the head of development, development teams were often too busy to deal with performance testing activities. Moreover, according to several informants, they often did not have the required technical knowledge to reliably perform them. Furthermore, many PRs referred the whole system architecture, hence they usually depended from the simultaneous development activities of multiple teams.

Another problem discussed was the decreasing number of performance tests executed over last releases (see Fig. 2.1). Following the example of other organizations [3], the researcher proposed to reserve part of the sprint to non-functional assurance activities (such as software performance assessment), in order to mitigate the increasing prevalence of functional activities over non-functional ones. Nevertheless, the proposal was not accepted by the head of development and product owners.

The lack of continuous performance assessment was also faced. The head of development proposed to automate PR execution to enable continuous performance testing of PRs. A product owner and the *ChogoRi* scrum master raised several practical concerns about this proposal. According to them, the assessment of some PRs implied the configuration of complex environments, which can be difficult to automate. Moreover, these tests often required days for configuration (e.g., database population) and executions on multiple dedicated machines, which were often not available in the organization. The researcher proposed to classify PRs based on their complexity, and to execute lightweight and easily automatable tests more frequently (e.g., every night), and complex tests with minor frequency. Similarly, the quality assurance specialist highlighted the need of identifying a subset of automated test that could continuously provide performance feedback against software evolution.

Another point discussed was the role of the PR list in the organization. The head of the development and the agile coach perceived the PR list as one of the major source of troubles in the performance assessment process. According to the quality assurance specialist, many PRs were probably outdated and related to old versions of the product. During a meeting, the researcher showed that 35 over 92 PRs were never tested in the last 6 releases and many others were only tested few times (see Fig. 2.4), thereby raising the concerns of the head of development and product owners about the utility of some PRs. To deal with this problem, a revision of the PR list was proposed as a potential first step.

After these meetings, the head of development decided to prioritize two main activities for the improvement of the performance assessment process: (i) the identification of a subset of automated performance tests and (ii) a revision of the PR list.

In order to perform the first activity, a meeting was held involving all architects and product owners to identify a subset PRs for automation. Three main criteria were used to identify this subset: the relevance of system operation under test, the ease of automation, the time effort required to configure and run tests, and the robustness of tests in terms of maintenance. At the end of the process, 12 PRs that exercise architecturally relevant operations were selected, involving 7 web test PRs and 5 load tests PRs. UI and mixed tests were discarded due to their poor robustness in terms of maintenance.

The second activity was performed in a meeting involving all product owners and software architects. Architects were typically reluctant to remove PR, since they aimed to carefully assess every aspect of the performance of the system. However, many PRs were rarely tested due to time and resource constraints. During the process, architects were invited to reflect on the potential drawbacks, in terms of management, of having a large list of (rarely tested) PRs. Eventually, 5 PRs were deprecated in the revision process.

2.5 Findings

Through a qualitative analysis of the data collected during the ethnographic study, we distill three key challenges in improving the performance assessment process in ASD, which we discuss in the following.

Managing performance assessment activities. The organization struggled to design a well-defined process to manage performance assessment activities. Potential solutions, borrowed from current practices, were discussed to improve the management of performance assessment activities, such as the use of PRs as constraints of backlog items[10] , or the use of the Definition of Done[11] [3, 14]. Nevertheless, these approaches were considered unsuitable for the organization. A challenge observed was the difference, in terms of goals and characteristics, of different performance assessment activities, which made difficult to design a unique process that works properly for all of them. For instance, some performance testing tasks were inherently linked to development activities (*e.g.,* assessment of new features), while others aimed to continuously monitor

[10] Nonfunctional Requirements - Scaled Agile Framework. https://bit.ly/3ohrZun

[11] Definition of Done, Agile Alliance. http://bit.ly/2YbdkWS

performance of architecturally relevant operations against system evolution. However, while the assessment of the latter was potentially crucial for any future software version, the relevance of the former was typically associated to a one-time development activity. Nevertheless, the organization managed both these performance activities in the same way, *i.e.,* they were both treated as permanent requirements. Therefore, a PR added with a high priority, for the assessment of a new feature, might remain highly prioritized over time due to the lack of priority updates (see Section 2.3.2), thereby leading to a suboptimal selection of performance assessment activities for future software releases.

Further studies are needed to design novel management approaches, which takes into account the differential nature of performance assessment activities.

Continuous performance assessment. Continuous assessment of software performance usually implies test automation. The choice of proper tests suite for automation and proper frequency of execution is crucial for successful performance assessment. The main challenge, in this regard, is to provide an adequate tradeoff between accuracy and frequency. Indeed, accurate tests usually involves complex configurations of production-like environments and long execution times, hence they cannot be frequently executed. On the other, lightweight tests are easily automatable and can be executed more frequently, but they are less representative of the real system usage, hence they have lower chances of detecting performance bugs. The organization relied on the knowledge of software architects to identify a subset of automated tests. However, nowadays, there is still little knowledge on how to properly tackle tradeoff between accuracy and frequency in performance assessment. More empirical studies are needed to fill this gap; for example, it would be valuable to understand to what extent lightweight and easily automatable performance tests (*e.g.,* microbenchmarks [76]) can mitigate the risks of performance failures.

Performance testing automation also implies to consistently establish whether a test is passed or not [43]. This problem was discussed in the organization, but no available solution was found. Although this problem was partially addressed through the use of targets in PRs, the identification of a performance bug often requires a thorough analysis of operational data that goes beyond the simple target check. Additionally, many PRs did not involve a target (*i.e.,* benchmarks), hence, in these cases, an automated technique is required to provide verdicts. The techniques proposed by Daly *et al.* [35], Nguyen *et al.* [97] and Chen *et al.* [25] seem promising in this regard. Nevertheless, further research is needed on this topic.

Defining the performance assessment effort. Another key challenge is the definition of the proper work effort devoted to performance assessment activities. The definition of a clear and commonly accepted balance between performance assessment activities and other development activities is essential to enable a reliable performance assurance process. Indeed, different actors in the organization had different perceptions on the relevance of performance assessment. Hence, when defining the performance assessment effort, it is crucial to consider these different viewpoints. For example, during the PR revision, software architects considered every single PR essential to enable reliable performance assessment. On the other hand, the head of development and product owners tended to de-prioritize these types activities during software development, thereby allocating resources that were not sufficient to perform the amount of performance assessment activities expected by architects. These conflicting viewpoints and the lack of a commonly shared vision on the amount of effort devoted to performance assessment led to an increasing number of (rarely tested) PRs, and caused several issues to the performance assessment process. Further empirical studies are needed to investigate whether this is common problem in ASD, and to identify potential best practices.

2.6 Threats to validity

2.6.1 Construct validity

Research involving people observation, such as ethnographic studies, may originate issues in terms of bias and rigor. Empirical research in industrial practice puts the researcher in a situation influenced by contradicting interests, hierarchies, and personal antagonisms [40]. To mitigate such issues, we first sought to establish a prolonged involvement in the fieldwork by keeping close contact with the organization members for six months. The development of a trusting relationship helped us to collect data from different perspectives, and also to observe the organization actors working in different Scrum ceremonies, meetings and daily activities. We also had access to system and process documentation. The significant amount of gathered data supported data triangulation which gave us a better confidence in our interpretation. In order to guarantee the quality of our descriptions, we took into account data gathered from different sources, *e.g.,* photographs, digital copies of documents and hand-written notes.

Following the best practices in ethnographic research [116], we didn't rely on a rigid plan, instead we keep data collection plans and expectations flexible to keep an "open mind" about the practices under study. Although this flexibility may suggest a lack of rigor, it is also considered as one of the main strengths of ethnographic research [46].

2.6.2 Internal validity

In our study, we were interested in understanding the practices of performance assurance as practitioners were interested in improving such practices. This might have influenced the participants' interaction with the researcher, as practitioners often expect recommendations and improvements when engaging with researchers conducting an ethnographic study [116]. Indeed, the researcher could be perceived as a managerial agent who will provide recommendations to improve their process. On the other hand, practitioners might hide aspects of their practice they do not want to have documented for management. In order to minimize this threat, we tried to be explicit about the intention of our research and the interaction with the involved stakeholders.

In the last three months of the study, the researcher actively participated to the improvement of the performance assurance process, thereby potentially influencing the outcome of the research. Although this could be a potential threat, the intent of our ethnographic study is to provide not only an in-depth understanding of the performance assurance practices, but also to support the improvement of such practices. Indeed, simply understanding practice may not be enough to satisfy the purpose of our research, as we aim to also understand the challenges in improving the performance assurance process. Additionally, when performing an ethnographic study, practitioners typically expect help by the researcher with improving their practices, and they could be puzzled if the researcher does not help to improve them [116].

The results obtained with ethnography tends to have a high internal validity [116] as far as the situation or context within which the evidence is gathered is consistent with the aim of the study [88]. Indeed, we conducted an ethnographic study in a R&D division of a large company that adopts Scrum [107], *i.e.,* a widely popular Agile development process, as we aim to understand the practices and the challenges for software performance assurance in the context of ASD.

2.6.3 External validity

There could be a limitation by focusing on one organization. Indeed, ethnographic studies are often criticized to have a weak external validity, *i.e.,* do not necessarily generalize to other contexts [88]. For example, the practices adopted by other organizations may be different and more effective. However, we compared the problems and challenges found in our study with those reported by other non-functional quality assurance studies in Agile contexts (*e.g.,* [3, 13, 69, 14]). The consistency of our results with the ones presented in other studies gives us confidence that the main findings of this study are portable to other organizations.

2.7 Related work

There are few studies concerning performance assurance in the context of ASD. Ho *et al.* [27] proposed an evolutionary model for PRs specifications, called PREM. PREM provides guidelines on the level of detail needed in a PR for development teams to specify the necessary performance details and the form of validation for the PR. Our case organization used an approach somehow similar to PREM, where PRs were first roughly defined by product owners (or architects), and then iteratively refined with the testing team. The same authors of PREM, in another article [65], described an experience in incorporating performance tests in Test-Driven Development. In our study, the researcher proposed to integrate performance testing activities as part of the development cycle. Nevertheless, the proposal was rejected since development teams were considered too busy and unexperienced to deal with this kind of activities.

Studies investigating non-functional quality assurance in the context ASD are related to our work. In an early study, Ramesh *et al.* [102] identified inadequate attention given to non-functional requirements as a major issue in ASD, by reporting specific concerns on software performance. The same problem was also reported in a systematic literature review on requirement engineering practices in ASD [61]. Similarly, in a recent empirical study, Kasauli *et al.* [69] reported difficulty in handling NFRs in the context of large-scale ASD, highlighting a lack of proper solutions to effectively handle NFRs. Interestingly, they reported that, similarly to our case organization, one of their case companies was facing problems in handling and keeping updated the list of NFRs. On the other hand, they also reported a reluctance to record NFRs in Agile companies, while we didn't observe a similar behavior in our case organization.

Behutiye *et al.* [13] identified four top categories of challenges for NFR management in ASD: the limited ability of ASD to handle NFR, time constraints due to short iteration cycles, limitations regarding the testing of NFRs and neglect of NFRs were the top categories of challenges. According to them, short iteration cycles and time constraints minimize the focus on addressing NFRs, and emphasize more implementation of functional features. In another study, Behutiye *et al.* [14] investigated documentation of NFRs in ASD. They found that different practices are used to document NFRs:

user stories, Definition of Done, acceptance criteria, documents, artifacts, prototypes and also face-to-face communication. According to them, companies approach to documentation of NFRs to fit the needs of their context, and the choice of the practices is affected by factors such as product domain, organization size and practitioners' experience.

Through a systematic literature review, Alsaqaf *et al.* [2] identified challenges that harm the management of NFRs in large-scale distributed Agile projects. Among them, there are: the inability of user stories to document NFRs, and the validation of NFRs that occurs too late in the process. The same researchers performed an empirical study [3] to identify mechanisms behind the challenges of managing NFRs in large-scale distributed Agile projects, and practices used to mitigate the impact of these challenges. They found that minimal documentation might result in missing the rationale behind NFR tradeoffs and architecture decisions taken earlier. Moreover, according to their findings, the priorities associated with user stories could be shuffled when deadlines are approaching and the need to deliver functional features becomes more critical than verifying NFRs. Example of practices adopted to mitigate challenges involves: reserving part of the sprint for NFRs assessment and the use of multiple product backlogs to include requirements of different viewpoints. Other studies address quality assurance in ASD in the specific context of safety-critical systems [57, 47].

Prior work broadly investigate non-functional quality assurance in ASD, while our study specifically target performance assurance. Moreover, these studies present results based on data collected through interviews and surveys, but do not examine how agile teams tackle non-functional quality assurance in their daily work. Instead, we explored how performance assurance is integrated in software development daily work by means of an ethnographic qualitative approach [116].

2.8 Conclusion

Our research objective was to explore practice and challenges for performance assurance in ASD. The study on the case organization highlighted difficulties in identifying a proper performance assessment process for ASD. Most of the current practices still rely on a waterfall-like methodology, which resulted in inconsistencies and poor performance assurance. Further research is needed to improve performance assurance in ASD. In this regard, we envisaged research directions related to three key challenges: (i) management approaches for performance assessment activities which take into account their diverse nature, (ii) tradeoffs and the current limitations in automated performance assessment, and (iii) defining the performance assurance effort.

Chapter 3

How Software Refactoring Impacts Execution Time

This chapter investigates the impact of refactoring on software performance. This work was conducted in collaboration with Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto and Vittorio Cortellessa. The content of this chapter is based on a submitted article [129], which is currently under review.

3.1 Introduction

Software systems are continuously changed to meet new requirements, fix defects, and enhance existing features. A key point for sustainable software evolution is high-quality source code. Indeed, several empirical studies have provided evidence that low code quality hinders maintenance and evolution activities [92, 70]. Tools have been developed to recommend to developers how to improve code quality via refactoring operations (*i.e.,* refactoring recommender systems) [10].

Despite their benefits, most tools ignore the heterogeneity of modern software, and the different priorities that non-functional requirements (*e.g.,* maintainability, performance, security) may have in different contexts. For example, smartphones have limited battery life and require software optimized to reduce energy consumption, while embedded systems often come with performance-critical requirements specifying precise time windows in which a task must be executed.

State-of-the-art refactoring recommenders target the improvement of code quality from a narrow perspective, focusing on improving code readability or removing well-known anti-patterns or code smells [131, 90, 11]. Basically, they *aim at improving* *code maintainability without considering the possible side effects that the recommended refactorings may have on other, maybe more important, non-functional requirements.* In other words, they do not consider the priority that different non-functional requirements may have. For this reason, some researchers started investigating the impact of "maintainability-driven" refactorings on other non-functional attributes.

Sahin *et al.* [108] showed that refactoring can change the amount of energy used by a software application, while Demeyer [38] investigated the impact on performance of introducing virtual functions in C++ code. These studies started laying the empirical foundations for building more *sensible* refactoring recommender systems, able to consider trade-offs between multiple non-functional requirements when making recommendations. The only concrete example is the EARMO tool by Morales *et al.* [91], able to support the refactoring of mobile apps by removing anti-patterns while controlling for the energy efficiency of the app. There is still a lack of empirical knowledge about the impact of refactoring on non-functional requirements.

We present a comprehensive study to investigate the impact of 16 different types of refactoring on the execution time of 20 Java systems. The systems have been selected given their attention to execution time, demonstrated by the presence of performance benchmarks in their code repositories. Using RefMiner [133] we mined the subject systems for "refactoring commits", *i.e.,* commits which contain refactoring operations. Each refactoring commit is accompanied by the code components (in our study, methods or classes) impacted by the refactoring. We manually inspected each commit to ensure that refactoring was its only goal. Through dynamic code analysis we identified the code components executed by the performance benchmarks in each system, and the refactoring operations that impacted them. Overall, we collected 82 commits implementing 167 refactoring operations impacting performance-relevant components. Each commit provides several data points for our study, since refactorings implemented in the same commit can impact different performance-relevant components and, thus, exercise different performance benchmarks. The total number of data points involved in our study (*i.e.,* pairs of refactoring actions, benchmarks) is 1,598. The collection of this data required ∼476 machine days. Besides presenting quantitative results showing the impact of (different types of) refactoring on execution time, we also qualitatively analyze cases in which refactoring had a negative impact on execution time, distilling lessons learned useful to (i) developers, for avoiding specific refactoring scenarios when performance is key, and (ii) researchers, for developing performance-aware refactoring recommenders.

Our results show that refactoring can have a substantial impact on execution time, both in a positive and negative way. Moreover, certain types of refactorings are more prone to degrade execution time and should be carefully performed in performance-critical systems.

3.2 Study Design

The *goal* of the study is to investigate the impact of refactoring operations on software performance. Measuring performance encompasses multiple metrics, such as response time, utilization, etc. In the context of this work, we focus on execution time, intended as the time that a section of code needs to be executed, without any concurrency and/or resource sharing with other software running on the same platform.

The *context* is represented by 82 commits mined from 20 systems in which developers performed a total of 167 refactoring operations impacting code components exercised by performance benchmarks. The study answers the following research questions (RQs):

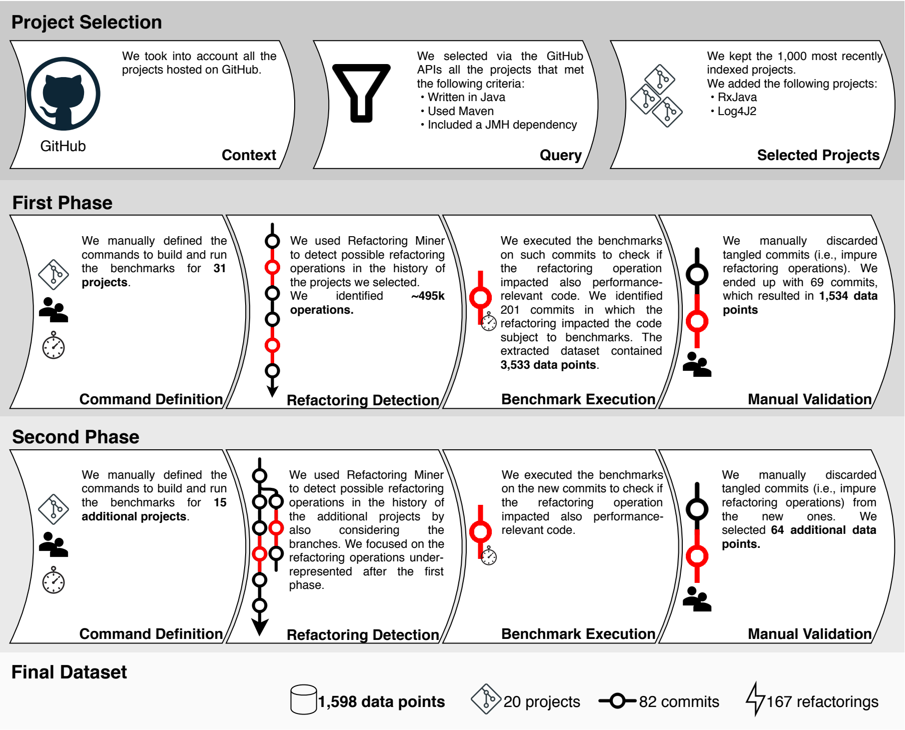
RQ$_1$ *To what extent do developers refactor performance-relevant code components?* We want to understand if developers are reluctant to refactor performance-relevant parts of the system.

RQ$_2$ *What is the impact of refactoring on performance?* We analyze the relationship between refactoring and performance, computing the percentage of cases in which refactoring improved, deteriorated, or did not impact performance.

RQ$_3$ *What types of refactoring operations are more likely to impact performance?* We investigate the relationship between types of refactoring operations (*e.g.,* Extract Method) and performance. Besides quantitatively analyzing our findings, we report interesting examples in which the refactoring had a negative impact on performance and we distill lessons learned useful for both researchers and practitioners.

3.2.1 Data Collection

We describe the procedure we followed to collect the data needed for our RQs. Specifically, we (i) selected Java open-source projects with performance benchmark suites, (ii) detected refactoring operations to assess their performance impact, and (iii) ran benchmarks before and after the refactoring operations were performed. We report in Fig. 3.1 an overview of the process we used to collect our data.



We took into account all the projects hosted on GitHub.

We selected via the GitHub APIs all the projects that met the following criteria:
• Written in Java
• Used Maven
• Included a JMH dependency

We kept the 1,000 most recently indexed projects.
We added the following projects:
• RxJava
• Log4J2

We manually defined the commands to build and run the benchmarks for **31 projects**.

We used Refactoring Miner to detect possible refactoring operations in the history of the projects we selected. We identified **~495k operations.**

We executed the benchmarks on such commits to check if the refactoring operation impacted also performance-relevant code. We identified 201 commits in which the refactoring impacted the code subject to benchmarks. The extracted dataset contained **3,533 data points**.

We manually discarded tangled commits (i.e., impure refactoring operations). We ended up with 69 commits, which resulted in **1,534 data points**

We manually defined the commands to build and run the benchmarks for **15 additional projects**.

We used Refactoring Miner to detect possible refactoring operations in the history of the additional projects by also considering the branches. We focused on the refactoring operations under-represented after the first phase.

We executed the benchmarks on the new commits to check if the refactoring operation impacted also performance-relevant code.

We manually discarded tangled commits (i.e., impure refactoring operations) from the new ones. We selected **64 additional data points.**

Fig. 3.1 Overview of our data collection process.

Project Selection

We selected projects in which developers defined micro-benchmarks for performance assessment. To do so, we queried GitHub for Java projects having a dependency with Java Microbenchmarking Harness (JMH)[1], the *de facto* standard for micro-benchmarks. While other micro-benchmarking tools are available (*e.g.,* Caliper, Japex, or JUnitPerf), they are either less popular than JMH, discontinued, or not executable in an automated way [120, 79].

We used the GitHub APIs to obtain the list of the 1,000 most recently indexed projects that (i) used Maven as the dependency manager (*i.e.,* they had at least a file named `pom.xml`), and (ii) had an explicit dependency with `org.openjdk.jmh.jmh-core`, *i.e.,* the core library required to run JMH. We considered only projects having at least 100 stars, and 88 projects satisfied this criterion. In addition to them, we considered two popular Java projects already used in a previous microbenchmark-related study [78], *i.e.,* RxJava and Log4j2. They were not included in the set of projects we initially selected because they were not among the 1,000 most recently indexed. Additionally, RxJava uses Gradle instead of Maven as a build tool.

[1] JMH, https://openjdk.java.net/projects/code-tools/jmh/

We manually analyzed the list of projects to find the commands that would build and run the benchmark suite. We were able to identify working commands and runnable benchmarks for 31 projects (including RxJava and Log4j2).

Refactorings and Benchmarks Gathering

We used RefactoringMiner [132] to extract refactoring operations performed on the default branch of each project. RefactoringMiner is able to detect 55 types of refactoring operations (*e.g.,* Extract Method, Extract Class, etc.). Given the time-consuming nature of our data collection, we discarded refactoring operations likely to have negligible or no performance impact. Specifically, we did not consider seven refactoring types: *Rename-related* refactoring operations (Rename Method, Rename Class, Rename Variable, Rename Parameter and Rename Attribute) and *package-related* refactoring operations (Change Package, Move Class).

Given a git repository of a Java project, RefactoringMiner reports (i) the commits featuring refactorings, (ii) the refactoring types, and (iii) the files and lines affected by the refactoring. We collected 494,826 refactoring operations (of 48 different types) performed in 181,020 commits and 31 projects. We identified benchmarks suitable to evaluate their performance impact, by verifying for each benchmark b in the project whether the code affected by refactoring operations is exercised by b. We first derived, for each refactoring r performed within the commit c, the set of Java methods impacted by r, namely M_r^c. We used srcML [29] to parse the Java files affected by r and we identified the set of methods M_r^c based on the impacted lines of code returned by RefactoringMiner.

For each benchmark b and for each commit c returned by RefactoringMiner, we derived the set M_b^c of Java methods executed by b. We built system snapshots both before and after the commit c is performed and ran dynamic analysis on the benchmarks to identify methods invoked by them. If we were not able to build one of the two snapshots or to run the benchmark suite, we discarded the commit c. We ran each benchmark b for 1 second and recorded the methods invoked in the execution using Java Flight Recorder (JFR)[2], to derive M_b^c. This required ∼221 machine days, involving the profiled execution of more than 4M benchmarks across 7,901 systems snapshots.

Finally, we intersected the list of methods affected by refactoring operations with those executed by benchmarks to identify benchmarks suitable to assess the performance impact of refactoring operations. We identified a set of 3,533 data points. Each data point is denoted by a tuple (p, c, b, R), where p indicates the project, c the commit, b

[2] JFR: https://tinyurl.com/y7yq8xc2

one of the project's benchmarks, and R a subset of refactoring operations performed in c. Each data point is such that every refactoring $r \in R$ modifies at least one method executed by b, *i.e.,* $M_r^c \cap M_b^c \neq \emptyset$. The 3,533 data points involved 201 commits, 521 refactoring operations (of 24 types) and 26 projects.

It is worth noting that a commit c may be a tangled commit [59], involving other code changes (apart from refactorings) which can affect parts of code executed by b. Including tangled commits might mislead the assessment of the impact on performance of refactoring operations. Therefore, we filtered out such cases by performing manual analysis on all the data points to verify whether refactoring operations are the only code changes in the commit that affect the code executed by b.

We grouped the data points by commit, and randomly assigned them to five of the researchers involved in the study, who manually analyzed the data points (p, c, b, R) assigned to them by inspecting the diff of the commit c, the commit message accompanying c, the set of methods invoked by the benchmark b (namely M_b^c), and the discussions in the issue tracker related to c (if a link to the discussions could be identified). The goal of the manual analysis was to decide whether b can be reliably used to evaluate the performance impact of the refactoring operations R (*i.e.,* no other interfering changes were impacting the methods exercised by b besides the refactoring operations). If a researcher classified a data point as relevant for our study (*i.e.,* a pure refactoring impacting b), it was double checked by another researcher. We only kept data points confirmed as relevant for our study by two of the researchers.

The dataset resulting from this process contains 1,534 data points involving 69 commits, 150 refactoring operations, and 16 refactoring types across 17 projects.

Benchmarks execution / Performance data collection

The performance comparison of different software versions in Java applications is far from trivial. There are a number of sources of non-determinism, such as Just-In-Time (JIT) compilation and optimizations in the Java Virtual Machine (JVM) [52]. We relied on steady state performance [52]: We repeated a benchmark execution for several *iterations* and collected measurements only after a steady state had been reached. We discarded measurements obtained in the first iterations, also called *warm-up iterations*, to avoid noise due to performance variations in transient states, usually caused by class loading and JIT (re)-compilation. Different VM invocations running multiple benchmark iterations may result in different steady-state performance data. For this reason, we also repeated benchmark iterations multiple times on different VM invocations.

JMH allows to define the number of warm-up iterations, measurements iterations and VM invocations directly in Java code or via command line arguments. We used the number of iterations defined in the code by benchmark developers for warm-up and measurement iterations, and we fixed the number of VM invocation to 10 (the JMH default) as done in previous studies [52, 77].

Given the previously described dataset, for each data point (p, c, b, R), we built system snapshots for the project p both before and after the commit c was performed, and we executed b on both snapshots, collecting two sets of measurements: E^{before} and E^{after}. Both of them are matrices, where $E_{i,j}$ represents the observed execution time for j-th benchmark iteration on the i-th VM invocation. Execution of benchmarks for the 1,598 data points of our study required 79 machine days on a dedicated machine.

Although all data points of our dataset are suitable to evaluate the performance impact of refactoring operations in general (see RQ$_2$), many of them (∼19%) cannot be used to analyze the relationship between types of refactoring operations and performance (RQ$_3$), since they involve multiple types of refactoring operations. We performed a second round of data collection specifically targeting the identification of commits in which a single type of refactoring was performed, focusing on data points (p, c, b, R) where all the refactoring operations $r \in R$ are of the same type.

To speed up the process, we removed 41 refactoring types from our data collection since, during the data collection performed for RQ$_2$, they had few related data points (<50). Also, we did not collect additional data points concerning Extract Method since we already had sufficient data points in our dataset (166 data points, 40 refactoring operations, 18 commits and 9 projects). As a result, the second round of data collection focused on six types of refactoring: Extract Superclass, Inline Variable, Extract Class, Move Method, Inline Method and Extract Interface.

For the supplementary data collection we mined refactorings (of the six targeted types) by launching RefactoringMiner on all the branches of all 31 projects. We also derived commands to run and build benchmarks for 15 additional projects gathered from [78]. Then, we derived (p, c, b, R) data points, as described in Section 3.2.1, but discarded those having multiple types of refactoring operations in R. Finally, performance measurements were collected for each data point as described in Section 3.2.1. Overall, we found 64 additional data points, which involve 17 refactoring operations, spanning 13 commits and 6 projects. The profiled execution of benchmarks to derive M_b^c sets lasted 176 machine days. The execution of the benchmarks took ∼11 machine hours.

This additional dataset is only used to analyze the relationship between types of refactoring operations and performance (RQ$_3$), while it is neither used to evaluate the performance impact of refactoring operations at coarse-grained level (see RQ$_2$), nor to evaluate the density of refactoring operations in parts of the system known to be performance-relevant (RQ$_1$). The inclusion of these data points may mislead the analysis of refactoring operations in general, since it could give more weight to the six refactoring types that are targeted in the supplementary data collection. We collected 1,598 data points, 167 refactoring operations of 16 types, 82 commits, over 20 projects.

3.2.2 Data Analysis

Answering RQ$_2$ and RQ$_3$ requires to determine whether refactoring operations have an effect (either positive or negative) on software performance. For this reason, we first describe the process we used to determine, for a given data point (p, c, b, R), whether refactoring operations R cause *regression*, *improvement* or *unchanged performance* in benchmark b.

Reliably detecting performance change

To determine whether refactoring operations lead to non-negligible performance change, we used the approach proposed by Kalibera & Jones to build confidence intervals for ratio of mean execution times [67, 68]. Compared to other performance change detection techniques (*e.g.,* hypothesis testing with Wilcoxon rank-sum combined with effect sizes [52, 77], and change-detection through testing for overlapping confidence intervals [52, 77]) the main benefit of the Kalibera & Jones technique is that, in addition to a reliable performance-change detection, it provides a clear and rigorous account of the performance change magnitude and of the uncertainty involved. For example, it can indicate that a system version is slower (or faster) than another by $X\% \pm Y\%$ with 95% confidence. To build the confidence interval we used bootstrapping [36], with hierarchical random re-sampling [104] and replacement. Re-sampling was applied on two levels [68]: VM invocations and iterations.

We ran 1,000 bootstrap iterations. At each iteration, new realizations of E^{before} and E^{after} measurements were simulated and the relative performance change was computed. The simulation of the $\hat{E}^{before}$ new realizations randomly selected a subset of real data from E^{before} with replacement. Similarly $\hat{E}^{after}$ was simulated by randomly sampling E^{after}. The two means and the relative performance change for simulated measurements were computed as follows:

$$\mu^{before} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\hat{E}_{i,j}^{before}}{mn} \quad \text{and} \quad \mu^{after} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\hat{E}_{i,j}^{after}}{mn} \quad \text{and} \quad \rho = \frac{\mu^{after} - \mu^{before}}{\mu^{before}}$$

After the termination of all iterations, we collected a set of simulated realizations of the relative performance change $P = \{\rho_i \mid 1 \leq i \leq 1000\}$ and estimated the 0.025 and 0.975 quantiles on it, for a 95% confidence interval. Given a (p, c, b, R) data point, refactoring operations R lead to a *regression* of b, if the lower limit of the confidence interval for relative performance change of mean execution times is greater than 0 (*i.e.,* b becomes slower after the commit c). Similarly, there is an *improvement* in b if the upper limit of the confidence interval is less than 0 (*i.e.,* b is faster before the commit). Otherwise, we consider performance as *unchanged.*

RQ$_1$: To what extent do developers refactor performance-relevant code components?

Performance benchmarks usually cover only parts of the system that are relevant to performance [76]. To answer RQ$_1$, we compared the density of refactoring operations in performance-relevant code to the one in other parts of the system. We considered commits where at least one refactoring was detected and for which we were able to collect methods executed by benchmarks suites. Table 3.1 reports, for each project, the number of commits and refactoring operations considered in this RQ. For each commit c we computed:

- PM_c: the number of performance-relevant methods (*i.e.,* methods executed by at least one benchmark) subject to at least one refactoring.
- NPM_c: the number of performance-relevant methods not subject to any refactoring operation.
- OM_c: the number of methods in the project not executed by any benchmark and subject to at least one refactoring.
- NOM_c: the number of methods in the project not executed by any benchmark and not subject to any refactoring.

Then, we computed, for every subject system, *refactoring density* in performance-relevant code as the ratio of the number of performance-relevant methods subject to

Project	Analyzed Commits (total)	Refactorings (total)	Methods (average)	Performance-relevant Methods (average)
alibaba/fastjson	34	85	1,249	21
apache/arrow	38	434	2,470	192
apache/camel	327	6,059	53,440	316
apache/commons-bcel	25	98	2,551	204
apache/logging-log4j2	405	1,700	2,161	300
arnaudroger/SimpleFlatMapper	80	1,291	3,148	170
cantaloupe-project/cantaloupe	204	1,297	1,767	121
debezium/debezium	82	417	2,843	107
easymock/objenesis	10	108	88	11
eclipse-ee4j/jersey	28	435	9,338	351
eclipse-vertx/vert.x	139	1,377	4,071	96
eclipse/jetty.project	392	2,218	12,400	86
eclipse/rdf4j	35	240	11,004	737
elastic/apm-agent-java	85	328	1,162	63
HdrHistogram/HdrHistogram	10	55	593	52
iotaledger/iri	29	329	1,220	50
JCTools/JCTools	75	740	870	95
jdbi/jdbi	31	107	1,282	170
jooby-project/jooby	59	425	1,546	7
kiegroup/drools	118	885	26,909	220
netty/netty	88	482	13,223	1,102
OpenFeign/feign	12	50	457	107
OpenHFT/Chronicle-Core	1	1	154	3
openzipkin/zipkin	15	165	1,698	249
panda-lang/panda	145	1,538	1,758	56
prestodb/presto	810	9,558	25,527	464
prometheus/client_java	6	13	264	35
protostuff/protostuff	11	93	1,994	35
pwm-project/pwm	24	560	4,551	6
ReactiveX/RxJava	16	225	3,915	943
zalando/logbook	9	103	513	99

Table 3.1 RQ$_1$. Number of commits and refactoring operations used to evaluate density of refactoring operations. The table also reports (for these commits) the average number of methods in the system and the average number of methods covered by at least one performance benchmark.

refactoring operations over the total number performance-relevant methods in the entire system history:

$$RDP_C = \frac{\sum_{c \in C} PM_c}{\sum_{c \in C} PM_c + NPM_c}$$

where C is the set of commits under analysis for the subject system. Similarly, we measured refactoring density in code not considered as performance-relevant:

$$RDNP_C = \frac{\sum_{c \in C} OM_c}{\sum_{c \in C} OM_c + NOM_c}$$

It is worth noting that we may count several times the same method if it appears in different snapshots of the system. For example, given a system with two commits,

c^1 and c^2, let us consider a performance-relevant method m subject to at least a refactoring operation in both c^1 and c^2.

Such a method is counted both in PM_{c^1} and in PM_{c^2}, *i.e.,* it is counted twice in the formula of RDP_C. We do this because the same method can have different properties in different commits: it could be counted as PM_{c^1} in a snapshot and as $NPM_{c^2}/OM_{c^2}/NOM_{c^2}$ in another one.

Finally, we report, for systems with more than 50 commits, refactoring density in performance-relevant code as well as refactoring density in other parts of the system via bar charts (Fig. 3.2). We also perform Fisher's exact test [117] to test whether the proportion of $\sum_{c \in C} PM_c / \sum_{c \in C} NPM_c$ and $\sum_{c \in C} OM_c / \sum_{c \in C} NOM_c$ differ significantly.

In addition, we used the Odds Ratio (OR) [117] of the two proportions as effect size measure. An OR of 1 indicates that refactoring performance-relevant code is equally likely as refactoring other parts of the system. An OR greater than 1 indicates that refactoring operations are more likely performed in code non-relevant from a performance perspective. An OR lower than 1 indicates that refactorings are more likely performed in performance-relevant code.

RQ$_2$: What is the impact of refactoring on performance?

Concerning RQ$_2$, we need to assess the impact of refactoring-related commits on software performance. In this RQ, we consider the dataset gathered from the first round of data collection, which involves 1,534 data points, 69 commits, 150 refactoring operations (among 16 refactoring types) and 17 projects (see Table 3.2 for an overview of the study-subjects projects).

Each refactoring-related commit has one of the following effects on the system performance:

- *regression*: the commit leads to performance regression of some benchmarks without improving performance of any other benchmark;
- *improvement*: the commit leads to performance improvement of some benchmarks without worsening performance of any other benchmark;
- *mixed*: the commit leads to performance regression of some benchmarks and improves the performance of some other benchmark;
- *unchanged*: the commit keeps the benchmark execution time unmodified.

Project	Stars	Commits	Benchmarks
alibaba/fastjson	22,752	3,793	4
apache/arrow	6,684	8,065	47
apache/camel	3,524	49,254	23
apache/logging-log4j2	1,075	11,238	585
cantaloupe-project/cantaloupe	172	4,376	11
eclipse/rdf4j*	229	4,279	132
eclipse-vertx/vert.x	11,552	4,825	41
hazelcast/hazelcast**	4,079	30,670	144
HdrHistogram/HdrHistogram	1,786	740	75
iotaledger/iri	1,183	2,701	3
JCTools/JCTools	2,518	971	172
jgrapht/jgrapht**	1,802	3,185	91
kiegroup/drools	3,356	12,894	1
netty/netty	25,443	10,100	1,686
OpenFeign/feign*	6,471	857	13
prestodb/presto	11,454	18,431	1,534
protostuff/protostuff	1,550	1,580	31
raphw/byte-buddy**	3,904	5,383	39
ReactiveX/RxJava	43,867	5,810	1,302
zalando/logbook	733	1,626	20

Table 3.2 RQ_2 & RQ_3. Overview of projects considered in the evaluation of the performance impact of refactoring operations (*i.e.,* for which at least one data point was discovered). Projects marked with (*) are only considered in RQ_2, those marked with (**) are only considered in RQ_3.

We report the percentages of refactoring-related commits falling in the four above categories via bar charts in Fig. 3.3. We also report the percentages of data points showing regressions, improvements or unchanged performance, to evaluate how code affected by refactoring operations is impacted in terms of software performance. In order to provide a comprehensive view on the performance impact of refactoring operations, we also report the magnitude of the performance change for benchmarks showing *regression* and *improvement.* The magnitude of performance change, for a given data point, is measured using the estimated mean relative performance change (*i.e.,* the center of confidence interval, see Section 3.2.2). We depict the distribution of these means via box plots for both data points showing regressions and data points showing improvements in Fig. 3.5.

RQ_3: What types of refactoring operations are more likely to impact performance?

To answer RQ_3, we need to isolate the effect of different refactoring types on software performance. We selected from our dataset the data points (p, c, b, R) having all refactoring operations $r \in R$ of the same type. Types of refactoring with less than 50 associated data points are excluded from this analysis. We analyzed 1,156 data points from 18 systems (see Table 3.2) involving 7 refactoring types: Extract Method (166 data points), Extract Superclass (90), Inline Variable (65), Extract Class (398), Move Method (184), Inline Method (66), and Extract Interface (187).

To analyze the impact on software performance of each refactoring type, we report via bar charts the percentage of data points showing regression, improvement and unchanged performance (see Fig. 3.6). We also report the magnitude of regressions and improvements for each type of refactoring via box plots (see Figures 3.7 and 3.8). In the latter analysis we discarded types having negligible impact both in terms of regression and improvement, *i.e.,* those that have less than 5% associated data points showing regressions or improvements. Finally, we discuss interesting examples related to different types of refactoring operations.

Qualitative analysis

To better understand how and why refactoring operations impact the performance, 5 of the researchers involved in the study manually inspected commits, issues and pull requests related to cases in which refactoring had a negative impact on execution time. We report interesting cases and discuss them along with RQ_2 and RQ_3 results.

3.2.3 Replication Package

We provide in our replication package [130] the complete data needed to replicate our findings. In particular, we share the SHA code of the subject commits from each of the analyzed systems, together with the refactoring operations detected in them and the results of the benchmarks execution. We also provide the *Python* scripts used to generate the figures and tables reported in Section 3.3.

3.3 Results Discussion

3.3.1 RQ_1: To what extent do developers refactor performance-relevant code components?

Fig. 3.2 reports the density of refactoring operations in performance relevant and non-performance relevant methods by software system.

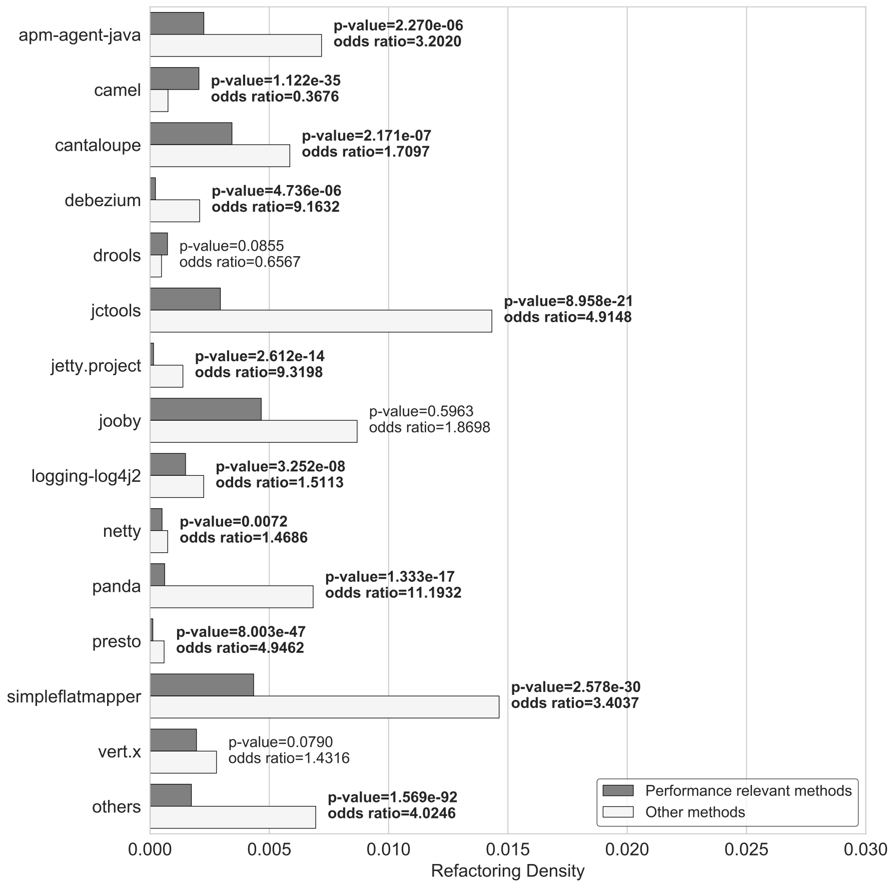


Fig. 3.2 RQ$_1$. Density of refactoring operations in performance relevant methods and in other methods. Fisher's exact test results accompanied with Odds Ratio are also reported.

As previously explained, we group all systems for which we collected less than 50 commits relevant for RQ$_1$ in "others" (bottom of Fig. 3.2). The first observation that can be made by looking at Fig. 3.2, is that no matter whether the method is performance relevant or not, the chance that it is subject to refactoring operations is quite low (*i.e.,* less than 1.5%).

When comparing the refactoring density in performance-relevant methods with that of performance-non-relevant methods, interesting trends can be observed. In only two projects (*i.e.,* `camel` and `drools`), developers performed more refactoring operations on performance-relevant methods. However, according to the Fishers's exact test, only in one project (*i.e.,* `camel`) such a difference is statistically significant (p-value < 0.05) with an OR of 0.37.

For all other projects, the refactoring density is higher in performance-non-relevant methods. Among those, only two projects (*i.e.,* `jooby` and `vert.x`) have a p-value larger than 0.05 (*i.e.,* the difference is not statistically significant). For all other projects, the refactoring density difference is statistically significant, with ORs varying from 1.47 to 11.19. This result indicates that in most projects, the density of refactoring operations is higher in non-performance relevant methods.

We conjecture that the potential performance impact might be one of the factors which discourage developers from refactoring performance-relevant methods. Validating such a conjecture would require a dedicated empirical study surveying developers. While this is out of the scope of this work, we proceed in the following RQs with investigating the impact on the execution time of the refactoring operations that focused on performance-relevant methods.

3.3.2 RQ$_2$: What is the impact of refactoring on performance?

Fig. 3.3 shows the impact of refactoring-related commits on execution time.

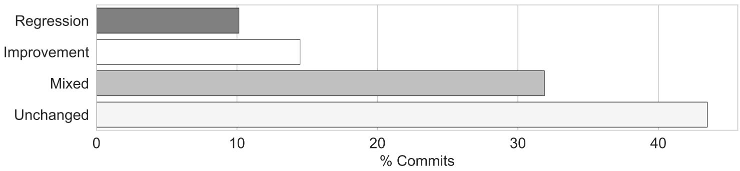


Fig. 3.3 RQ$_2$. Percentages of refactoring-related commits leading to regression, improvement, mixed effect or unchanged execution time.

It is worth noting that multiple benchmarks can be involved in each commit. As can be seen from the chart, more than 40% of refactoring-related commits do not result in any performance change. In the rest of the cases, a large percent of the commits (>30%) have a mixed effect on performance. That is, the performance improves in some involved benchmarks while regresses in others. Only around 15% of the commits lead to performance improvement and slightly more than 10% cause only performance regression. This is not surprising as code affected by refactoring operations can be exercised in different ways by benchmarks. For example, each benchmark may involve the execution of a different set of performance-relevant methods of the system or the execution of the same methods with different inputs, which can lead to diverse performance behaviors. From these results, we can observe that a large percent of the commits (>55%) causes a statistically significant performance change (either positive or negative) on methods of the system that are considered relevant for performance.

Specifically, when considering the negative impact on performance due to refactorings, more than 40% of the commits causes performance regression in at least one benchmark. This is particularly relevant considering that performance issues are usually discovered through specific tests and inputs [64, 81, 125].

By inspecting how the performance is impacted in each benchmark affected by refactoring-related commits (Fig. 3.4), we can find that in more than 75% of cases, the performance is not changed.

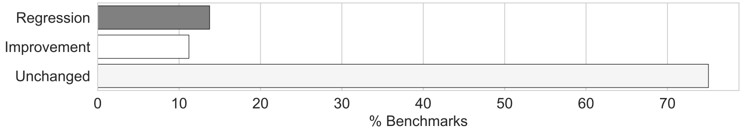


Fig. 3.4 RQ_2. Percentages of benchmarks affected by refactoring-related commits (*i.e.,* data points) showing regressions, improvements or unchanged execution time.

Neither performance regression nor performance improvement is common, and they both take place in around 10% of the benchmarks. This suggests that, similarly to common performance issues [64], performance changes introduced by refactoring operations require specific benchmarks to be exposed [81, 125]. That is, even when the commit causes a performance change in some of the benchmarks, there are often other benchmarks for which performance remains unchanged.

We further looked into the extent of performance regression or improvement (Fig. 3.5), and we can find that the medians of relative performance changes are below 5% for both performance improvement and regression.

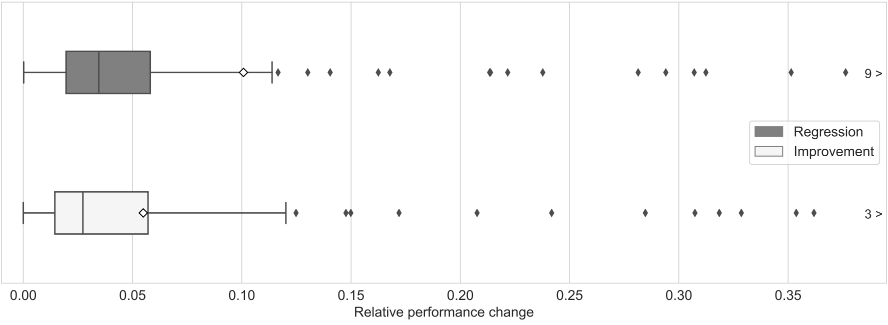


Fig. 3.5 RQ_2. Relative performance change of benchmarks due to refactoring operations. The darker box plot reports results for benchmarks showing regression, while the lighter box plot reports results for benchmarks showing improvement.

About 50% of regressions involve performance changes between 2% and 6%, and, similarly, 50% of improvements ranges between 1% and 6%. Moreover, it is rare that refactoring-related commits can lead to a performance change of more than 15%. This is expected considering that we are investigating code changes performed in a single code commits. The magnitude of these changes, which may appear negligible, is still relevant as benchmarks measure performance at method level [76–78]. Indeed, even a relatively small performance change at method level may potentially lead to a huge performance deviation at system level.

For example, in the pull request `8614`[3] of `netty/netty` we found that a code refactoring was not accepted as it causes "non-negligible" performance regression (*i.e.,* up to 5%) in two benchmarks. This confirms that performance changes due to refactoring operations (see Fig. 3.4) may be relevant for performance.

The pull request `8614`, mentioned above, also highlights interesting aspects about the relationships between refactoring activities and software performance. The goal of this pull request is to achieve "less code duplication" and "better encapsulation" by removing duplicated logic from two classes with the help of a common helper class, and it involves several refactoring operations such as Extract Class, Change Parameter Type, and Move Method. After the performance regressions were detected in the two benchmarks, the developer revised the code changes and eliminated the regression, and finally the pull request was merged. Nevertheless, while our benchmarks results show no-regression on the benchmarks used by developers (which is inline with developer expectations), we did find significant performance regressions (up to 3 times) on other benchmarks not considered by developers. This may suggest that *comprehensively analyzing the performance impact of refactoring operations is not trivial, and even experienced developers might consider only a part of it.*

Another interesting fact is that although some projects attach great importance to the performance, they merge refactoring-related commits without verifying their performance impact. For example, two commits (`49ac2da`[4] and `f537eda`[5]) performing "Extract Superclass" operations were proposed in the same pull request `185`[6] for the project `JCTools/JCTools`, in order to "homogenize atomic queues" (*i.e.,* making the atomic queue class `AtomicArrayQueue` as similar to the unsafe queue `ArrayQueue` as possible). While the author expressed concerns about performance ("Performance impact of this is unverified"), the pull request was merged without any discussion. Nevertheless, we found that these commits have non-negligible impact on performance (up to 11%). We conjecture that the long execution time required to run performance

[3] https://github.com/netty/netty/pull/8614

[4] http://bit.ly/3oRLfza

[5] http://bit.ly/34aqgjd

[6] https://github.com/JCTools/JCTools/pull/185

benchmark suites (*e.g.,* more than two hours for `JCTools/JCTools` [76]) may prevent developers from verifying the performance impact of refactoring operations. The adoption of state-of-the art techniques [78] to reduce benchmarks execution time without sacrificing result quality may be beneficial for this problem. Also, similarly to what has been done to predict the impact of a refactoring operation on quality metrics before applying it [22], it might be beneficial to design techniques able to predict the impact of refactoring operations on performance.

Summing up, most refactoring-related commits lead to performance change, with these changes usually affecting only a subset of the involved benchmarks. Moreover, we found that a large percent of commits (>55%) leads to regression in at least one benchmark. Performance regressions and improvements due to refactoring-related commits have relatively similar frequencies, and they can bring a performance change up to 12% in most of the cases. Nevertheless, these relatively small performance changes may still be relevant for system-level performance, especially in case of methods involved in "core features". Finally, our results indicate that the analysis of the performance impact of refactoring activities may be non trivial even for experienced developers, as these changes can have diverse (and often mixed) effects on performance-relevant methods. This problem is further exacerbated by the long execution time required to run benchmark suites, which may prevent developers from verifying the performance impact of their refactoring operations.

3.3.3 RQ$_3$: What types of refactoring operations are more likely to impact performance?

Fig. 3.6 reports the percentage of benchmarks in which the performance is positively or negatively impacted by each type of refactoring operation considered in RQ$_3$.

The chart reveals that all of the refactoring types can lead to both improved and regressed performance. Overall, Extract Class/Interface/Method/Superclass refactoring operations are more likely to impact performance than Inline Method/Variable and Move Method. When performing Extract Class/Interface/Method and Inline Method the performance is more likely to degrade, while when performing Inline Variable and Move Method, there is a higher chance of performance improvement. Extract Superclass leads to similar amounts of performance regression and improvement.

The Extract Class refactoring is the more closely related to performance regression, with more than 16% of impacting benchmarks showing such a trend. Moreover, the magnitude of regression introduced by Extract Class is higher when compared to other

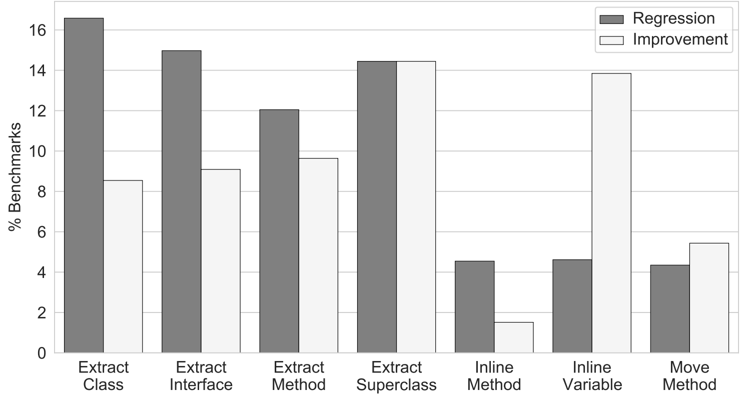


Fig. 3.6 RQ$_3$. Performance impact of different types of refactoring on the associated benchmarks (*i.e.,* data points). Percentages of benchmarks showing regression or improvement are reported for each refactoring type.

types of refactorings (50% of regressions lead to a performance change between 2% and 7%, see Fig. 3.7).

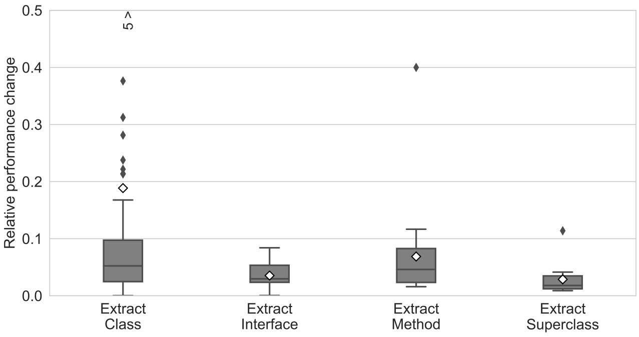


Fig. 3.7 RQ$_3$. Relative performance change of benchmarks showing regressions due to different types of refactoring operations.

Indeed, Extract Class refactoring may induce a higher number of allocated objects which may regress the performance of the system. For example, the description of the issue `LOG4J2-1295`[7] of `apache/logging-log4j2` states "*it is not always obvious that some code creates objects, so it is easy for regressions to creep in during maintenance code changes*". Indeed, according to `apache/logging-log4j2` developers[8] "*garbage collection (GC) pauses are a common cause of latency spikes*" and the allocation of more temporary objects "*contributes to pressure on the garbage collector and increases*

[7] https://issues.apache.org/jira/browse/LOG4J2-1295

[8] http://bit.ly/3apTONJ

the frequency with which GC pauses occur". Therefore, when Extract Class refactoring causes a higher number of allocated objects, it may increase the frequency of GC pauses, thereby leading to performance regression. Another interesting fact is that even the same Extract Class operation can have significantly different performance impact on slightly different versions of software. For example, when we inspected a case of non-negligible performance regression (*i.e.,* performance reduced 8% and 20% for two benchmarks, respectively) caused by Extract Class in the commit `90d82d2`[9] of `apache/logging-log4j2`, we found the same refactoring operation was performed in another branch of the same project (`9ad3603`[10]). However, this refactoring only causes regression of up to 5% for seven different benchmarks. This finding suggests that even the same Extract Class operation can have different impact on performance under different contexts.

Extract Interface, Extract Superclass and Extract Method also have higher chances to lead to performance regressions compared to other refactorings (respectively, ∼15%, ∼14% and ∼12% of the involved benchmarks show regression). While the former two cause less intense regressions, Extract Method provides regressions with similar magnitudes to those observed for Extract Class (see Fig. 3.7). Although improvements due to Extract Method are less frequent than regressions, they lead to higher performance changes (50% of them range from 2% to 14%, see Fig. 3.8).

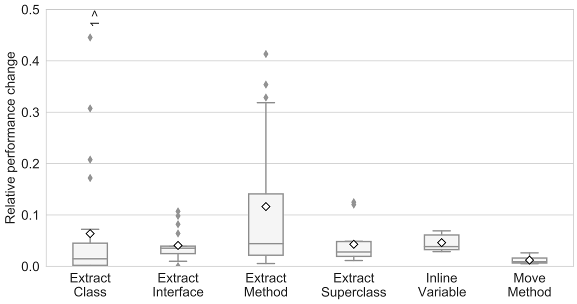


Fig. 3.8 RQ_3. Relative performance change of benchmarks due to different types of refactoring operations.

This behavior relies on the relationship between Extract Method and specific runtime optimizations employed by the JVM, as smaller methods have more chances to be inlined during runtime optimization of a Just-In-Time (JIT) compiler. Extract Method refactoring is common practice to achieve automatic inlining at runtime. It

[9] http://bit.ly/2WnBOv3
[10] http://bit.ly/38cVpnd

is often difficult to identify such optimization opportunities as they require specific conditions. To become a candidate for inlining, a method must satisfy at least one of two conditions: i) its bytecode size must be within 35 bytes (by default); or ii) it must be called more often than a pre-defined threshold (10,000 invocations by default) and its bytecode size must be within 325 bytes (by default) [98]. Usual benchmarks configurations are ineffective to identify such optimization opportunities. In commit `ceb0a62`[11] of `apache/logging-log4j2` which "*refactors a large method into smaller methods to enable inlining*", improvement is expected in execution time as these smaller methods have more chances to be inlined. Our benchmarks results only displayed performance regressions. This kind of optimizations may not manifest when evaluating performance through default benchmarking configurations, as they are triggered only in specific scenarios (the authors mentioned in the commit message that "*the new code is all inlined after ∼7000 invocations*"). We envision that future research may leverage static characteristics of the code (*e.g.,* size of methods) combined with its dynamic behavior (*e.g.,* the number method invocations during benchmarking) to design recommenders which automatically suggest potential optimization opportunities through Extract Method refactoring.

For the other types of refactoring, Inline Variable has a relatively high chance to lead to performance improvement (14% of the benchmarks with a performance change ranging from 3% and 6% in 50% of the cases), while Move Method and Inline Method have lower chances to bring performance change. Moreover, the performance change caused by the Move Method has never reached 5%.

In summary, the impact of refactoring on performance varies from type to type. No refactoring type guarantees the absence of performance regression. Extract Class and Extract Method have a higher chance of causing larger performance regression than other types of refactoring. When Extract Method causes performance improvement, it leads to larger performance changes.

3.4 Threats to Validity

Construct validity The main threats to the construct validity of our study are related to the process we adopted to measure performance variations caused by refactoring.

To mitigate the risk of unstable performance benchmark results, we perform, within each VM invocation, multiple warmup and measurement iterations accordingly to the JMH configuration defined by software developers, and we fix the number of VM

[11] http://bit.ly/3gXtiN2

invocations to 10 as done in previous studies [52, 77]. We did not use developer custom configurations for VM invocations, since a previous study [78] showed that developers often rely on a single VM invocation, which is considered a bad practice as inter-JVM-variability is common [52, 67, 76]. Using configurations with higher number of iterations or VM invocations may lead to more stable results. Prior work [52] suggests to dynamically stop measurement iterations when certain quality criteria are met (*e.g.,* coefficient of variation < 0.02). Nevertheless, we found this approach impractical for our study due to extremely long execution times. Software compilation may also induce performance variability [93] due to the non-deterministic nature of Java compilation strategies [53, 67]. Such variability can be mitigated through compiler replay [67, 53] to avoid bias introduced by compilation. However, these approaches can dramatically increase the time needed for benchmarking as they add another level of repetition, which makes them impractical for our study. To reliably detect performance change, while dealing with performance variability, we followed best practices [67, 77, 20, 78]. We estimate the confidence interval for relative performance change with bootstrap [68, 36] by employing hierarchical random resampling with replacement [104], and we detect performance change if there is statistically significant difference, *i.e.,* the confidence interval does not contain 0. Other techniques, such as A/A testing, can be used to further validate the robustness of the performance change identification framework [77], but we didn't rely on them given their extremely long execution times. Nevertheless, it is worth to notice that we minimized the chance of reporting false positives by estimating the confidence interval of relative performance change [67, 68]. Indeed, transient noises in the execution environment tend to enlarge the confidence interval, thereby increasing the chance of not reporting a performance change (*i.e.,* 0 falls in the confidence interval).

To answer RQ_1, we considered as "performance relevant" the code that is covered by at least a performance benchmark. There may be performance-sensitive parts of the system that are not covered by benchmarks. Nevertheless, the fact that a piece of software is covered by a performance benchmark means that developers consider its performance as worth to evaluate. Since the goal of our RQ is to verify whether developers are reluctant to refactor code that is relevant in term of performance, we consider such an assumption as acceptable for the sake of our study.

Imprecisions in the detected refactorings could also have affected our results. However, we used a highly precise state-of-the-art tool (RefactoringMiner [132]), reported to have a 98% precision and 87% recall. Also, while it is possible that we missed relevant data points for our study due to false negative (*i.e.,* refactoring-related commits missed by RefactoringMiner), we are confident about the absence of false positives in our dataset, since we manually inspected all the commits subject of our study to exclude those introducing, besides the detected refactorings, other code changes.

Conclusion validity. Wherever possible we used appropriate statistical procedures with p-value and effect size measures to test the significance of the differences and their magnitude.

Internal validity. Those are mainly related to a missing causation link between refactorings and changes in performance as assessed by the benchmarks, and to possible confounding factors that may influence such a relationship. We controlled for tangled commits, ensuring that the commits considered in our study only focused on refactoring-related changes. However, (i) in our observational study we do not claim causation, and (ii) at least, we complemented the quantitative analysis with a qualitative one, which helped in better understanding the influence of refactoring on performance.

External validity. While the number of systems and the subject commits is limited as compared to those of MSR studies, it is worth nothing that the data collection procedure for our study required 15 months of work. Moreover, the number of systems we consider is larger than recent studies investigating software performance research questions (see *e.g.,* [76, 78, 39, 103]), while the numbers of commits and performance tests involved are similar. Still, the generalizability of our findings is limited to the analyzed refactoring types and systems.

3.5 Related Work

Given the goal of our study, we discuss the literature related to studies investigating (i) software performance in the context of code evolution, and (ii) the impact of refactoring operations on quality attributes.

3.5.1 Empirical Studies relating Performance to Software Evolution

Performance analysis of running software systems has been tackled by different perspectives in the last few years (*e.g.,* through models at runtime [54, 5]). However, given the goal of our work, we focus here on the domain of empirical analyses, possibly supported by benchmark techniques.

Han *et al.* [56] have introduced StackMine, a tool exploiting stack traces to allow performance debugging of a considerable amount of data on Windows-based systems.

Our approach works at higher level of abstraction because, on the basis of traces generated through JMH microbenchmarks, we aim at identifying the (beneficial or degrading) effects of refactoring actions on software performance.

Sandoval *et al.* [114, 112, 113] have analyzed performance regression of different versions of applications in an object-oriented language and development environment named Pharo[12]. They have analyzed 19 different projects and a total of 1,125 different versions. The main differences between our approach and the one in [114, 112, 113] are: first, the context (*i.e.,* Java systems *vs* Pharo); second, we have exploited JMH as micro-benchmarking library, instead of building an in-house benchmark suite. Furthermore, we have analyzed 20 different open source projects by generating 1,598 data points.

Daly *et al.* [35] have presented mechanisms for detecting performance regression in an industrial project, *i.e.,* MongoDB. They automatically detect change-points variability to identify the commit causing a specific performance degradation event. Then, those labeled points have been manually checked to discard false positives. Our process starts from commits labeled with specific refactoring actions and, then, look at their effect on performance. Also, our study spans different projects.

Laaber *et al.* [78] have focused their study on reducing the required execution time of micro-benchmarking tests through a dynamic reconfiguration of JMH. They have defined three ways to detect when a test reaches the performance peak (*i.e.,* the steady-state) and then they apply their reconfigurations. In our study, it would be interesting to use the approach proposed by Laaber *et al.* with the aim of reducing the duration of our tests. However, we have decided to exploit the default JMH configurations (*i.e.,* the ones associated to the different commits) to be as compliant as possible with developers intents.

Chen *et al.* [23] have studied the influence of code changes on performance degradation in the context of the Python programming language. They have exploited unit tests, along with a profiler, to extract performance data. Our study design differs from the one by Chen *et al.*, since we target Java programming language and exploit micro-benchmarks, instead of unit tests, to extract performance data. Also, we focus on a specific type of code changes (*i.e.,* refactoring actions).

Reichelt *et al.* [103] have compared unit tests to discover performance regression between versions of nine long-lived Java open-source projects. The use such a corpus to infer performance variations through code changes. We rely on JMH instead of unit tests, because the former avoids JVM optimizations that may produce unreliable performance data.

[12] Pharo project: http://pharo.org

Ding *et al.* [39] have analyzed whether unit tests can be aimed at assessing performance. In particular, they have targeted two systems (*i.e.,* Cassandra[13] and Hadoop[14]), and they have extracted functional tests that can be performance-related by digging developers message backlogs. We have instead dug the GitHub corpus in order to extract projects equipped with JMH tests, and we have investigated the correlation between refactoring actions and performance degradation in 20 systems.

Table 3.3 lists the sizes of corpora of related works that empirically assess software performance. To avoid second-guessing, we only report such data for studies explicitly providing information about the number of subject projects and benchmarks.

Reference	Projects	Benchmarks
Sandoval *et al.* [114, 112, 113]	19	1,125
Laaber *et al.* [78]	10	2,164
Chen *et al.* [23]	8	1,268
Reichelt *et al.* [103]	9	105
Our study	20	1,598

Table 3.3 Comparison among corpora sizes.

To the best of our knowledge, the magnitude of our study is on par with, if not larger than, previous works empirically analyzing changes in performance caused by source code changes.

3.5.2 On the Impact of Refactoring on Code Quality Attributes

In recent years, many researchers have focused on how refactorings might impact the quality of software projects.

Moser *et al.* [92] conducted a case study on a project developed in an agile and close-to-industrial environment. The authors examined the code quality change after refactorings, with complexity and coupling metrics. They found that refactorings lead to simpler and less coupled code.

Szőke *et al.* [122] analyzed five software systems and measured the quality change over refactorings with a probabilistic quality model. With the 200 identified refactoring commits, the authors found that while single refactoring does not necessarily increase the software quality, its increase in local components and globally can be more evident when refactorings are applied in blocks.

[13] Cassandra: https://cassandra.apache.org/

[14] Hadoop: https://hadoop.apache.org/

Tavares *et al.* [124] applied 80 refactorings automatically generated by JDeodorant on seven open-source Java systems and investigated how refactoring impacts code smells. Their results indicate that while some code smells can be eliminated by refactoring as expected, there are also cases that refactorings introduce new bad smells.

Abid *et al.* [1] examined the impact of refactorings on both security and other quality attributes (*i.e.,* reusability, flexibility, understandability, functionality, extendibility, and effectiveness). By analyzing 30 open-source software projects, they found that while refactorings help to improve other quality attributes, the software tends to become less secure. This negative correlation need to be taken into account before refactoring software systems.

Lin *et al.* [80] inspected 1,448 refactoring operations from 619 Java projects to understand whether refactorings lead to more natural code, namely whether the source code becomes more repetitive and predictable. Their results indicate that this assumption does not always hold, and the impact on the code naturalness varies among different types of refactorings.

Sahin *et al.* [108] conducted an empirical study, involving 197 applications of six commonly-used refactorings, to investigate how refactorings affect the energy usage. Their results show that all the considered refactorings in the study can potentially impact the energy consumption, with a magnitude ranging from -4.6% to 7.5%. Verdecchia *et al.* [135] also looked into the same topic. They applied automatic refactoring on five different types of code smells in three open-source Java projects and collected energy consumption in a controlled environment. As a result, they found that in one project, refactoring significantly impacted the energy consumption.

To the best of our knowledge, not many studies have investigated the impact of refactoring operations on the performance of software systems. The most relevant work to ours is the study conducted by Demeyer [38], which inspected how a specific type of refactoring (*i.e.,* replacing conditionals by virtual function calls) impacts the performance of C++ programs. Their results show that this type of refactorings often leads to faster performance compared to their non-refactored counterparts. While this study is highly relevant to software performance, it only focuses on a specific type of refactoring operation.

3.6 Conclusion

We presented an empirical study aimed at investigating the impact of refactoring operations on execution time. To the best of our knowledge, this is the first work analyzing a wide set of refactoring types from the "performance perspective". As for any performance-related study, the collection of the data needed to answer our research questions posed major challenges and required hundreds of machine days.

The achieved results show that the impact of refactoring on execution time varies depending on the refactoring type, with none of them being 100% "safe" in ensuring that there is no performance regression. Some refactoring types, such as Extract Class and Extract Method, can result in substantial performance regression and, as such, should be carefully considered when refactoring performance-critical parts of a system.

Our findings, disclosing the potential side-effects of refactoring on execution time, pave the way to the development of (i) approaches to predict the impact on performance of planned refactoring operations before they are actually implemented in the system, and (ii) *sensible* refactoring recommender systems, able to consider trade-offs between multiple non-functional requirements when making recommendations. Our future agenda is driven by these two research directions.

Chapter 4

Automating Performance Issue Diagnosis in Service-based Systems

This chapter introduces automated performance diagnosis in service-based systems, and presents the concept of *Latency Degradation Pattern* (LDP), which forms the theoretical basis of the techniques presented in Chapters 5 and 6.

The chapter first provides background on the topic and presents the LDP concept. Then, it explains how automated LDPs detection can be integrated in a DevOps process, and it overviews related work on automated performance issue diagnosis.

4.1 Background

Modern high-tech companies deliver new software in production every day [45] and perceive this capability as a key competitive advantage. In order to support this fast-paced release cycle, IT organizations often employ several independent teams that are responsible "*from development to deploy*" [99] of loosely coupled independently deployable services. Unfortunately, frequent software releases often hamper the ability to deliver high quality software [106]. For example, widely used performance assurance techniques, like load testing [62], are often too time-consuming for these contexts. Also, given the complexity of these systems and their workloads [7], it's often unfeasible to proactively detect performance issues in a testing environment [134]. For these reasons, today, the diagnosis of performance issues in production is a fundamental capability for maintaining high-quality service-based systems.

Service owners are usually responsible and accountable for meeting Service Level Objectives (SLOs) on Key Performance Indicators (KPIs). Common SLOs related to software performance are requirements on specific KPIs such as resource utilization (*e.g.,* CPU, memory), throughput and latency. Software engineers and performance continuously monitor KPIs and execution traces on the run-time system to identify symptoms of potentially relevant performance issues that lead to SLOs violations. The truly identification of such symptoms is often critical: a request may involve several Remote Procedure Calls (RPC) and the number of performance traces and performance metrics to analyze can be huge. According to a recent study on microservice-based systems [138], software engineers spend days or even weeks to debug a software issue, and initial understanding, scoping and localization are among the most time-consuming phases during debugging. Although several techniques have been introduced to provide automation in diagnosing performance issues in service-based systems [28, 87, 72, 95, 75, 9], the reduction of the manual effort and the time needed is still critical.

Automated diagnosis techniques aim at identifying patterns in operational data that are correlated to system anomalous execution. The identification of such patterns provides three main benefits: (i) they provide evidences based on data on the existence of relevant performance issues, (ii) they reduce the amount of operational data to inspect, and (iii) they provide useful information to effectively localize and debug performance issues. These patterns are usually extracted from two different types of operations data: *time-series metrics* and *traces*. The formers are dynamic system information measured over intervals of time (e.g. service throughput, CPU consumption), while the latters contain data about causally related events of individual end-to-end requests (*e.g.,* nominal RPC execution time for a particular request). Given the recent advancements [85, 84, 109, 66] and the widespread adoption of distributed tracing [118, 82, 66], this thesis specifically target automated pattern detection in *traces*.

Few automated diagnosis techniques have been introduced to detect patterns in traces [24, 56, 75, 9] . Some of these techniques specifically focus of categorical traces attributes [24, 56]. For example, StackMine [56] specifically targets *function names* within callstack traces, while the technique proposed by Chen et *et al.* [24] targets categorical request trace attributes such as *host machine names*, *request types* and *thread ids*. Other techniques [75, 9] enable pattern detection also on continuous trace attributes (*e.g., execution time*). Krushevskaja and Sandler [75] proposed a pattern detection technique based on combinatorial algorithms (*i.e.,* dynamic programming and branch-and-bound). This technique requires a preliminary encoding step to transform continuous trace attributes to binary features, however, the authors did not suggest any automated approach to perform this task. The only other example is DeCaf [9] by

Bansal *et al.*, which leverage a random forest model and ranks predicates extracted by the model according to their correlation with performance degradation.

This thesis aims to tackle pattern detection in continuous traces attributes, in the context of a specific kind of SLO violation, namely *latency degradation.* In particular, we explicitly target *Remote Procedure Call* (RPC) *execution time* due to its relevance for the diagnosis of *latency issues* in service-based systems. Chapters 5 and 6 present two novels fully automated techniques to detect *RPCs execution time behaviors* correlated to latency degradations. In this thesis, we name these behaviors as *Latency Degradation Patterns.*

4.2 Latency Degradation Patterns

Services are often subject to SLO on request latency (e.g. time to load the homepage of a website). Usually, a SLO on latency defines a range of acceptable values, *i.e.,* $L \leq L_{SLO}$. In this thesis, we name the range of latency values that do not meet SLO expectations as the *targeted latency range*, *i.e.,* $L > L_{SLO}$.

A request to a service-based system often involves several RPCs. Each request is associated to a set of execution trace attributes (*i.e.,* RPC execution time). In this thesis, we denote a request r as an ordered sequence of trace attributes $r = (e_0, e_1, ..., e_m, L)$, where e_j represents the execution time of a specific RPC j triggered by the request.

Latency Degradation Patterns (LDPs) are patterns in RPCs execution times correlated with SLO violation. They can be represented as conjunctions of predicates over RPCs execution time. Conjunctions of predicates are used, instead of single predicates, because several software issues in service-based systems lie in the interaction of multiple RPCs [138] rather than being rooted in the internal implementation of individual RPCs. Moreover, a single predicate alone is often not sufficient to capture the patterns of SLO violations [28].

An informal example of LDP could be:

> The homepage latency exceeds L_{SLO} when $Auth$ execution time is greater than 30 milliseconds and $getProfile$ execution time is between 20 and 50 milliseconds.

More formally, a pattern P is denoted as a set of predicates $\{p_0, p_1, ..., p_k\}$ with $k \geq 0$. A request r satisfies ($\lhd$) a pattern P if every predicate $p \in P$ is satisfied by the request r:

$$r \lhd P \iff \forall p \in P, \quad r \lhd p$$

Each predicate targets a specific RPC j and is denoted as a triple $p = \langle j, e_{min}, e_{max} \rangle$, where $[e_{min}, e_{max})$ represents a range of values on the RPC execution time. We say that a request $r = (..., e_j, ...)$ satisfies p, denoted as $r \lhd p$, if:

$$e_{min} \leq e_j < e_{max}$$

A previous approach use F1-score [75] to measure the degree of correlation between patterns and latency degradation. The idea is to partition the set R of requests under analysis in two subsets R_{pos} and R_{neg}, namely the set of requests not meeting SLO (or positives) and the set of requests meeting SLO (or negatives)

$$\begin{aligned} R_{pos} &= \{r \in R \mid L > L_{SLO}\} \\ R_{neg} &= \{r \in R \mid L \leq L_{SLO}\} \end{aligned} \tag{4.1}$$

and to compute F1-score for a pattern P accordingly:

$$F1\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{4.2}$$

where precision and recall of the pattern are defined as follows:

$$precision = \frac{\mid tp \mid}{\mid tp \mid + \mid fp \mid} \tag{4.3}$$

$$recall = \frac{\mid tp \mid}{\mid R_{pos} \mid} \tag{4.4}$$

and true positives tp and false positives fp are defined as:

$$\begin{aligned} tp &= \{r \in R_{pos} \mid r \lhd P\} \\ fp &= \{r \in R_{neg} \mid r \lhd P\} \end{aligned} \tag{4.5}$$

If a pattern shows high recall then it frequently appears in requests with latency falling in the *targeted latency range.* But, it does not provide any guarantees on its infrequency in requests meeting SLO. On the other hand, a high value of precision indicates that most of the requests satisfied by the pattern do not meet SLO expectations, but the number of the involved requests may be negligible and not worth to investigate. F1-score, which is the harmonic mean of precision and recall, provides a unique measure to evaluate the quality of a pattern while keeping into consideration both these aspects.

4.3 Automated LDPs detection in a DevOps context

Automated approaches for LDPs detection can be easily integrated into a DevOps process, since they can be executed periodically (*e.g.,* every day or week) to detect symptoms of relevant performance issues in RPCs execution time. The only assumption is the presence of a distributed tracing infrastructure [109] (*e.g.,* Zipkin[1] , Jaeger[2], Dapper [118]), which, given the recent widespread adoption of distributed tracing solutions [83], seems a reasonable assumption for a modern service-based system.

Following the example of DeCaf [9], patterns derived by these techniques can be stored into a database along with their F1-scores (see Equation (4.2)) to build historical knowledge and to automatically diagnose different categories of potential performance issues:

1. *New*: A new LDP is identified that has never appeared in the past.
2. *Regressed*: F1-score of the LDP is substantially increased compared to the past.
3. *Known*: F1-score of the LDP is similar to the recent one.
4. *Improved*: F1-score of the LDP is substantially decreased compared to the past.
5. *Resolved*: LDPs that were previously detected do not appear anymore.

The description of each category is intentionally broad. Their concrete definition highly depends on the context (e.g. system characteristics and service owners needs), hence they are reported to provide the intuition on how these techniques can be integrated into a DevOps process.

4.4 Related work

In this section, we summarize prior work on automated diagnosis techniques based on operational data. These techniques can be classified into two broad categories: (1)

[1] https://zipkin.io
[2] https://www.jaegertracing.io

those that detect patterns in time-series metrics and (2) those that detect patterns in traces.

Cohen *et al.* [28] devised the first technique for automated diagnosis of performance issues in software systems. They used a class of probabilistic models (Tree-Augmented Bayesian Networks) to identify combinations of time-series metrics and threshold values that correlate with compliance with SLOs for average-case response time. Duan *et al.* introduced Fa [41], an automated diagnosis technique that uses anomaly-based clustering to clusters time-series metrics based on how they differ from those related to failure and pinpoints metrics linked to failure. MonitorRank [72] uses the historical and current time-series metrics, along with the call graph of the service-based system to build an unsupervised model for ranking. This technique identifies metrics correlated to system anomalies by using an adaptation of the PageRank algorithm [100]. Farshchi *et al.* [44] adopts regression-based analysis to find the correlation between operation's activity logs and the operation activity's effect on cloud resources. Other techniques for detecting patterns in time series metrics relies on association rule mining [17], hierarchical detectors [95], pairwise-correlation analysis [87] or clustering combined with correlation analysis [121].

The first work that falls in the second category (i.e, the one based on traces) is the one introduced by Chen *et al.*, which uses decision trees [24] to identify causes of failures. In this technique, decision trees are trained on traces, and combinations of trace attributes are ranked according to their degree of correlation with failure. Han *et al.* introduced StackMine [56], a technique that mines callstack traces to help performance analysts to effectively discover costly callstack patterns. StackMine identifies callstack patterns correlated with poor performance by using an adaptation of a classic association rule mining algorithm [136]. Unfortunately these techniques [24, 56] are unsuitable to identify patterns in continuous attributes (e.g. execution time), since they specifically target categorical trace attributes. For example, StackMine specifically targets function names within callstack traces, while the technique proposed by Chen *et al.* [24] targets categorical request trace attributes such as host machine names, request types and thread ids. At the best of our knowledge, the first automated diagnosis technique suitable for pattern detection in continuous trace attributes is the one proposed by Krushevskaja and Sandler [75]. In this technique, the pattern detection problem is modeled as as a binary optimization problem and solved using combinatorial search algorithms (i.e., dynamic programming combined with branch-and-bound algorithm or forward feature selection). Although this approach works with continuous trace attributes, a non-automated encoding step is required to transform continuous trace attributes to binary features. Recently, Bansal *et al.* introduced DeCaf [9], a technique based on random forests, which can be applied both on categorical and continuous attributes. Similarly to [24], this technique first trains a random forest model and then ranks predicates extracted by the model according to their correlation with system anomalies. Bansal *et al.* demonstrated that DeCaf can be applied on traces with categorical attributes with up to 1M cardinality, by evaluating their approach in two large scale services.

In this thesis we specifically focus on Latency Degradation Patterns (*i.e.,* RPC execution time patterns correlated with latency degradation), therefore we compared the effectiveness and efficiency of the presented techniques to those of techniques suitable to this problem, i.e. automated diagnosis techniques that can be applied for pattern detection in continuous trace attributes [75, 9].

Other studies on software diagnosis rely on visualization techniques. Beschastnikh *et al.* [15] introduced ShiViz, which presents distributed system executions as interactive time-space diagrams to help diagnosis and debugging of software issues. Zhou *et al.* [138] used ShiViz to conduct an empirical study to investigate the effectiveness of existing industrial debugging practices compared to those of state-of-the-art tracing and visualization techniques for distributed systems, thus showing that the current industrial practices of debugging can be improved by employing proper tracing and visualization techniques. The study of Sambavisan *et al.* [110] compares three well-known visualization approaches in the context of presenting the results of one automated performance root cause analysis approach [111]. Visualization techniques are time consuming, as they require human intervention and are more useful when performing fine-grained analysis, while the techniques presented in this thesis automatically detect patterns in RPCs execution times to identify potential relevant performance issues.

The techniques presented in this thesis detect patterns in traces collected by a distributed tracing infrastructure, therefore distributed tracing research [109] is related to our work. Dapper [118] was the pioneering work in this space, Canopy [66] processes traces in real-time, derives user-specified features, and outputs performance datasets that aggregate across billions of requests. Pivot Tracing [85] gives users the ability to define traced metrics at runtime, even when crossing component or machine boundaries. Related to our work are also studies on automated log parsing as they extract run-time operational data which can be then exploited by automated diagnosis techniques. These techniques focus on analyzing raw service logs to extract meaningful events and run-time information. The approach proposed by Jiang *et al.* [63] leverages clone detection techniques to uncover common tokens in log lines. LogCluster [94] proposed an approach that automatically parses log messages by mining the frequent tokens in the log messages. Logram [34] leverages n-gram dictionaries to achieve efficient log parsing, while other techniques formulated log parsing as a clustering problem and used various approaches to measure the similarity/distance between two log messages [51, 123, 55]. A systematic literature review on automated log parsing can be found elsewhere [42].

Chapter 5

LagranDe: Latency Degradation Pattern Detection in Service-based Systems

This chapter presents *LagranDe* (*Latency Degradation Pattern Detection*), a fully automated LDPs detection technique. Section 5.1 describes how *LagranDe* models the problem of detecting LDPs. Section 5.2 describes the *LagranDe* approach and its main components: dynamic programming algorithm, genetic algorithm and employed optimizations techniques. The research questions, experimental method, threats to validity and results are described in Section 5.3, while final remarks are presented in Section 5.4.

This work was conducted in collaboration with Vittorio Cortellessa and was published in the *11th ACM/SPEC on International Conference on Performance Engineering* [33].

5.1 Problem modeling

Intuitively, the problem of detecting LDPs can be modeled as an optimization problem, where the set of feasible solutions is represented by every feasible pattern P, and the objective function is the maximization of the pattern F1-score, as defined in Equation (4.2) in Chapter 4.

However, the identification of a single LDP may not be enough to "explain" the entire *targeted latency interval* (*i.e.,* $L > L_{SLO}$). Indeed, performance deviations in live service-based systems may be related to multiple simultaneous causes, which can

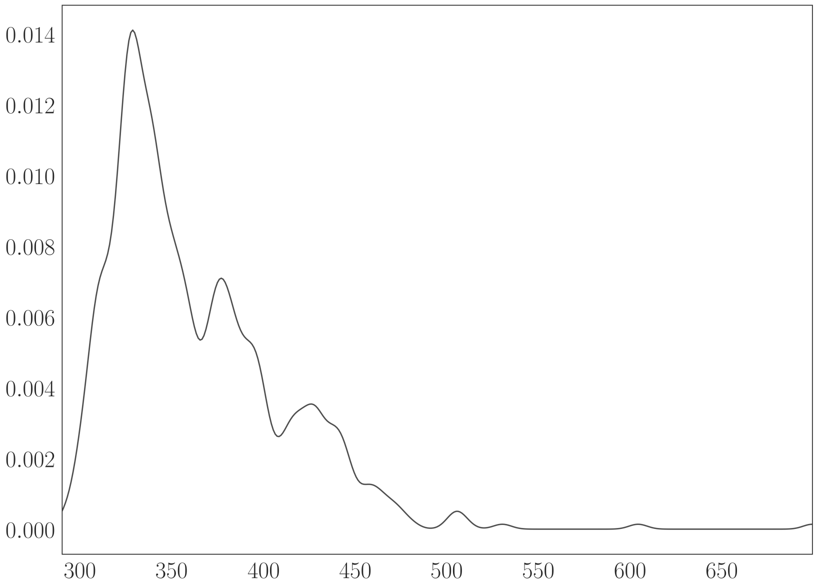


Fig. 5.1 Example of latency distribution.

lead to distinct LDPs. For this reason, LagranDe aims at identifying multiple LDPs for the *targeted latency interval*.

Different performance issues often lead to different performance behaviors, and, therefore, to multiple "modes" in the request latency distribution[1] (see Fig. 5.1). Following this intuition, LagranDe splits the *targeted latency interval* in sub-intervals, and searches for each sub-interval the optimal pattern in terms of F1-score.

In order to describe how LagranDe models the problem of detecting LDPs, in the following we adapt the concepts of precision, recall and F1-score described in Chapter 4, to a generic sub-interval I (of the *targeted latency interval*). First, the sets of positive and negative requests are redefined as follows:

$$\begin{aligned} R_{pos} &= \{r \in R \mid L \in I\} \\ R_{neg} &= \{r \in R \mid L \notin I\} \end{aligned} \tag{5.1}$$

Then, the F1-score of a pattern P with respect to a sub-interval I can be computed accordingly.

$$Q(P, I) = 2\frac{precision \cdot recall}{precision + recall} \tag{5.2}$$

[1] Frequency Trails: Modes and Modality. https://bit.ly/2Ziyzqb

where precision and recall are defined as

$$\begin{aligned} precision =& \frac{\mid tp \mid}{\mid tp \mid + \mid fp \mid} \\ recall =& \frac{\mid tp \mid}{\mid R_{pos} \mid} \end{aligned} \tag{5.3}$$

and the sets of true positives and false positives for a given pattern P are defined as

$$\begin{aligned} tp &= \{r \in R_{pos} \mid r \lhd P\} \\ fp &= \{r \in R_{neg} \mid r \lhd P\} \end{aligned} \tag{5.4}$$

The optimal pattern P^* for a given sub-interval I can be found by searching the pattern that maximizes the F1-score:

$$P^* = \arg\max_P Q(P, I) \tag{5.5}$$

Obviously, a sub-optimal partition of the *targeted latency interval* may hamper the effective detection of LDPs. For this reason, we use the approach proposed in Krushevskaja and Sandler [75] to effectively partition the *targeted latency interval*. In this approach, a set of potential split points $\{s_0, s_1, ..., s_k\}$ is pre-defined on the *targeted latency interval*, where $s_0 = L_{SLO}$ and s_k is the maximum observed request latency. This set can be derived using local minima in latency distribution to partition groups of requests that show similar latency behavior, hence that can be potentially associated to different performance issues.

Let $\Theta(s_i, s_j) = \max_P Q(P, (s_i, s_j))$ be the score of a sub-interval (s_i, s_j). The ultimate goal of our approach is to identify a subset of split points $\{s_0^*, s_1^*, ..., s_z^*\}$ (where $z \leq k$, $s_0^* \equiv s_0$ and $s_z^* \equiv s_k$) that maximize the following equation:

$$\sum_{i=0}^{z-1} \Theta(s_i^*, s_{i+1}^*) \tag{5.6}$$

The key intuition is that by optimizing the sum of scores, the *targeted latency interval* is partitioned in sub-intervals in a way that favors the identification of relevant patterns.

5.2 The LagranDe Approach

The main problem of maximizing Equation (5.6) requires the solution of the sub-problem described by the Equation (5.5). In order to solve the main problem we use the dynamic programming approach proposed in [75]. Let $D(i)$ denote the best score for a solution that covers interval $[s_0, s_i)$, with the initial score D(0) = 0. The update step is:

$$D(i) = \max_{0 \leq j < i} (D(j) + \Theta(s_j, s_i)) \tag{5.7}$$

Hence, to construct the solution that covers interval $[s_0, s_i)$ the algorithm search for each possible pair i, j such that $D(j) + \Theta(s_j, s_i)$ is maximized.

In the following we describe how we solve the sub-problem, which represents the core novelty of our approach. We first describe the components of our search-based approach, and then we describe how fitness evaluation is optimized through search space reduction and precomputation.

5.2.1 Genetic algorithm

Our approach uses Search Based Software Engineering (SBSE) [58], an approach in which software engineering problems are reformulated as search problems within the search space that can be explored using computational search algorithms. Specifically, we use a Genetic Algorithm (GA). GAs are based on the mechanism of the natural selection [60] and they use stochastic search techniques to generate solutions to optimization problems. The advantage of GA is in having multiple individuals evolve in parallel to explore a large search space of possible solutions.

Our GA is implemented on top of the DEAP framework [48]. In the following we describe the five key ingredients of our GA implementation (i.e., representation, mutation, crossover, fitness function and computational search algorithm) in the context of our sub-problem defined in Equation (5.5).

Representation: Feasible solutions to the sub-problem are all possible patterns P. We recall that a pattern is defined as a set of predicates $P = \{p_1, p_2, ..., p_k\}$, where each predicate is a triple $< j, e_{min}, e_{max} >$, with j referring to the RPC subject to the predicated and $[e_{min}, e_{max})$ representing the execution time interval.

Mutation: The mutation randomly choose among three mutation actions, namely: *add*, *remove* or *modify*. The first action adds a new randomly generated predicate to the pattern P. However, if the RPC involved in the new predicate is already present in another predicate contained in P then the mutation doesn't have any effect.

The remove mutation randomly removes a predicate from P. The modify mutation randomly selects a predicate p and modifies one of the two endpoints of the interval, and then reorders them, if necessary, so that $e_{min} < e_{max}$.

Crossover: Given two patterns P_1 and P_2, the two sets of predicates are joined together $P_U = P_1 \cup P_2$ and then randomly partitioned in two new patterns P_1' and P_2'.

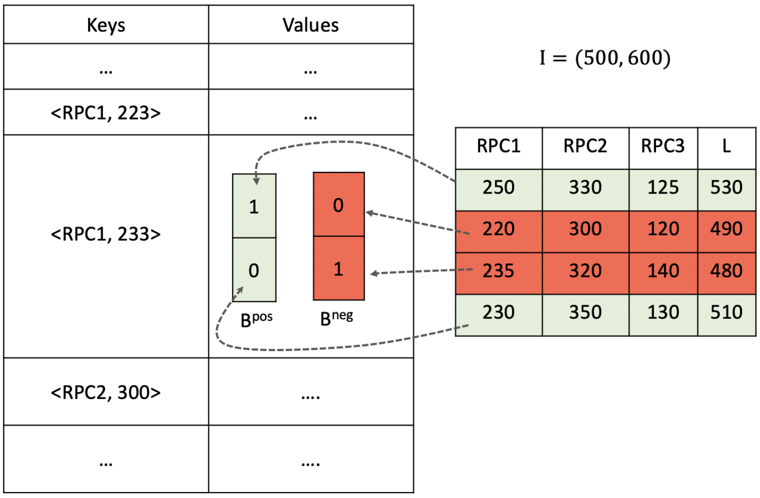
Fitness function: In order to evaluate the fitness of each solution, we adopt the quality score described by Equation (5.2). However, the computation of such score can be overwhelmingly expensive, hence we optimize the fitness evaluation with the techniques described in next subsection.

Computational search: We use a $(\mu + \lambda)$ genetic algorithm [16]. As a selection operator, we use a tournament selector [74] with tournament size by 20. Crossover and mutation rates are fixed to 0.8 and 0.2, whereas μ and λ are both set to 100. The evolutionary process terminates after 400 generations with a 100 population size.

5.2.2 Optimization of fitness evaluation

Fitness evaluation is one of the most frequently executed operations during the evolutionary process. A time-consuming fitness evaluation can severally hamper the approach efficiency. The presented approach uses precomputation to enhance fitness evaluation performance, where the employed technique requires a search space reduction. Although search space reduction can potentially cut off optimal or near-optimal solutions, we employ a smart reduction policy which still preserves search space quality. In the following we explain the key idea behind our precomputation technique, then we illustrate the search space reduction policy.

Precomputation: The identification of true positives and false positives for a given pattern P is the most performance-critical operation executed during fitness evaluation. This operation requires to verify for each $r \in R$ if $r \lhd P$. The verification of this property involves a bunch of *inequality checks*, which are likely to be repeated several times during the evolution process. The aim of our technique is to reduce fitness evaluation effort by precomputing *inequality check* results in order to avoid redundant computations. We denote inequality checks as pairs $\langle j, e_t \rangle$, where j is a RPC and e_t is an execution time threshold. Inequality check results are represented as ordered sequences of booleans $B = \langle b_0, b_1, ..., b_n \rangle$, where b_i refers to the check result for the request $r_i \in R$. A check result b_i for a given inequality check $\langle j, e_t \rangle$ and a request



Keys	Values
...	...
<RPC1, 223>	...
<RPC1, 233>	B^{pos}: 1, 0 B^{neg}: 0, 1
<RPC2, 300>	....
...	....

I = (500, 600)

RPC1	RPC2	RPC3	L
250	330	125	530
220	300	120	490
235	320	140	480
230	350	130	510

Fig. 5.2 Hash table entry for inequality check <RPC1, 233> over a set of requests and an interval I=(500, 600)

$r_i = (..., e_j, ...)$ is defined as:

$$b_i = \begin{cases} True & \text{if } e_j \geq e_t \\ False & \text{otherwise} \end{cases}$$

Since the aim is to both improve true positives and false positives computation, inequality check results are precomputed on positive and negative requests (i.e., R_{pos} and R_{neg}). Hence, for each inequality check $\langle j, e_t \rangle$ two boolean sequences are generated (namely B^{pos} and B^{neg}), which represent inequality check results, respectively, for positive and negative requests. Boolean sequences are encoded as bit strings and stored in a hash table, where the key is an inequality check $\langle j, e_t \rangle$ and the value is a pair of bit strings $\langle B^{pos}, B^{neg} \rangle$. Figure 5.2 shows a simplified example of a hash table entry for an inequality check over a set of requests and interval I.

These data structures enable fast identification of true positives and false positives across multiple requests through bitwise operations. A bitwise operation works on one or more bit strings at the level of their individual bits. We use two common bitwise operators: *and* ($\wedge$) and *not* ($\neg$) .

A predicate $p = \langle j, e_{min}, e_{max} \rangle$, can be efficiently evaluated on positive requests as well as on negative requests with the following two steps. First, boolean sequences associated to inequality checks $\langle j, e_{min} \rangle$ and $\langle j, e_{max} \rangle$ are retrieved from the hash table. We denote them as $\langle B^{pos}_{min}, B^{neg}_{min} \rangle$ and $\langle B^{pos}_{max}, B^{neg}_{max} \rangle$ respectively. Then, positive and negative requests satisfying predicate p are derived through bitwise operations:

$$B_p^{pos} = B_{min}^{pos} \wedge \neg B_{max}^{pos}$$

$$B_p^{neg} = B_{min}^{neg} \wedge \neg B_{max}^{neg}$$

Where $b_i \in B_p^{pos}$ (resp. $b_i \in B_p^{neg}$) denotes if $r_i \lhd p$ with $r_i \in R_{pos}$ (resp. $r_i \in R_{neg}$). The same approach is also applied for pattern satisfaction, $r_i \lhd P$:

$$B_P^{pos} = \bigwedge_{p \in P} B_p^{pos}$$
$$B_P^{neg} = \bigwedge_{p \in P} B_p^{neg}$$

Number of true positives and false positives are then obtained by counting $True$ booleans (i.e. number of 1 in the bit string) in both B_P^{pos} and B_P^{neg}:

$$|tp| = |\{b \in B_P^{pos} \mid b = True\}|$$
$$|fp| = |\{b \in B_P^{neg} \mid b = True\}|$$

Finally, fitness is derived through a simple numerical computation (see Equations (5.3) and (5.2)).

Search space reduction: Since the execution time is a continuous value, there is an uncountable number of possible inequality checks $\langle j, e_t \rangle$. Hence, precomputing results for any possible inequality check $\langle j, e_t \rangle$ is unfeasible. For this goal, we employ a search space reduction to decrease precomputation effort as well as the amount of inequality check results (i.e. bit strings) to store. Obviously, search space reduction can be risky, since optimal or near-optimal solutions can be excluded by the search. We tackle this problem by selecting, for each RPC, only meaningful thresholds e_t according to execution time distribution. We select thresholds that separate high density regions of the execution time interval. Those values identify relevant points that should cluster together requests with similar execution time behavior in a certain RPC. The key intuition is that if execution time of a certain RPC interval is correlated with a relevant latency request degradation, then its behavior must be recurrent to a relevant number of requests (hence, to a dense interval of the execution time). Furthermore, RPC execution time distribution often shows multimodal behavior and modes can be often related to performance degradation (e.g., cache hit/miss, slow/fast queries, synchronous/asynchronous I/O, expensive code paths). Our approach employs a mean shift algorithm [30] to identify high density intervals of RPC execution time. Mean shift is a non-parametric feature-space analysis technique for locating the maxima of a density function [26], and its application domains include cluster analysis in computer vision and image processing [30]. For each RPC, we cluster requests with the mean shift algorithm according to the corresponding execution time, we then infer thresholds according to identified highly dense regions.

5.3 Evaluation

In this section, we state our research questions (RQs) and we present the evaluation of our approach on a microservice-based application case study. We chosen microservices because they represent a widely used paradigm in nowadays service-based systems. In addition we envision that our approach can be extremely useful to debug performance issues in microservice-based applications, given high frequency of deployments and continuous experimentation [115](e.g., canary and blue/green deployment).

5.3.1 Research questions

We aim at addressing the following research questions:

RQ1 Is our approach effective for clustering requests associated to the same latency degradation pattern, as compared to machine learning algorithms?

In order to answer this question, we compare our approach against three general-purpose machine learning clustering algorithms (*i.e.,* K-means, Hierarchical, Mean Shift) that are described in Section 5.3.5. The rationale beyond this question is the widespread of modern machine learning tools and libraries for clustering problems.

RQ2 Is our approach effective with respect to state-of-the-art approaches for latency profile analysis?

With this respect, we have identified the work in [75] as the closest one to our approach, also described in Section 5.3.5. The differences are that they adopt a branch and bound algorithm and they target a more general problem, because the attributes that they consider are not limited to latencies. We have implemented their approach for sake of result comparison.

RQ3 How robust is our approach to "noise"?

We have introduced two types of *noise* in our experiments, which are described in Section 5.3.3 and can affect detection capabilities of our approach. We have compared the above mentioned approaches in terms of their effectiveness in presence of noise.

RQ4 What is the efficiency of our approach as compared to other ones? This question strictly concerns the execution time. We have measured the execution time of all considered approaches applied on the case study, for sake of a costs/benefits analysis.

5.3.2 Subject application

We experiment our approach on E-Shopper[2], that is an e-commerce microservices-based web application. The application is developed as a suite of small services, each running in its own Docker[3] container and communicating via RESTful HTTP APIs. It is composed by 9 microservices developed on top of the Spring Cloud[4] framework, where each microservice has its own MariaDB[5] database. The application produces traced data that are reported and collected by Zipkin[6], *i.e.,* a popular distributed tracing system, and stored in Elasticsearch[7]. Traced information are then processed and transformed in a tabular format, such as the one showed in Table 5.1.

getProfile execution time (ms)	getRecommended execution time (ms)	getCart execution time (ms)	...	Request latency (ms)
200	60	60	...	320
280	75	90	...	450
...	...	...	...	...
220	60	60	...	390

Table 5.1 Tabular representation of traces

Each trace refers to a single request, while trace attributes are RPC pure execution time. By pure execution time we mean the RPC execution time minus the time waiting for invoked RPCs to terminate. For example, in Figure 5.3 the RPC `getHome` calls three synchronous RPCs (`getProfile`, `getRecommended`, `getCart`), hence pure execution time of getHome is its execution time minus the time waiting for termination of the three invoked RPCs. Note that asynchronous calls do not introduce waiting time, thus we do not consider them in pure execution time calculation. We focus our analysis only on requests loading the homepage, since it is the request which trigger more RPCs. Specifically, each request involves 13 calls of 8 unique RPCs among 5

[2] https://github.com/SEALABQualityGroup/E-Shopper

[3] https://www.docker.com

[4] https://spring.io/projects/spring-cloud

[5] https://mariadb.org/

[6] https://zipkin.io

[7] https://www.elastic.co

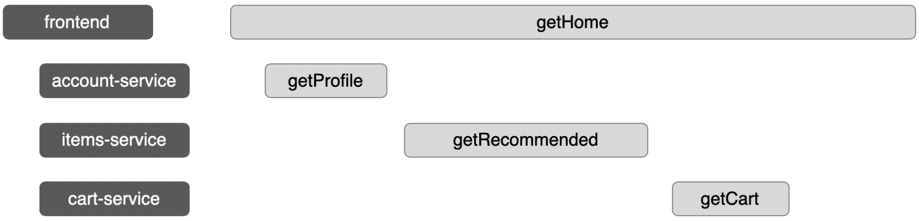


Fig. 5.3 Gantt chart example

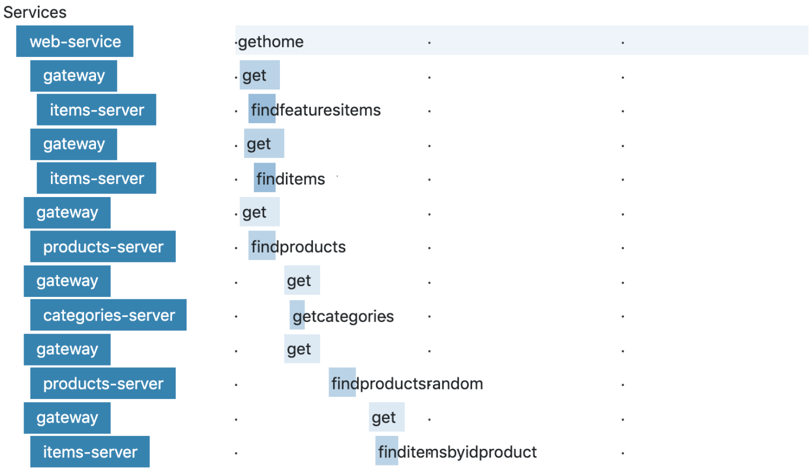


Fig. 5.4 Gantt chart of a request loading the homepage of E-Shopper

microservices, as showed by the Gantt chart showed in Figure 5.4. Note that the first two `get` RPCs invoked by `web-service` are asynchronous, hence they do not block `gethome` execution.

5.3.3 Methodology

The main goal of our empirical study is to determine whether the presented approach is able to identify clusters of requests affected by the same degradation causes. In order to achieve this goal, we perform multiple load test sessions in which we inject recurrent *artificial degradations*, thereafter we run our approach on each set of collected traces to evaluate whether clusters of requests affected by same artificial degradations are correctly identified. In our empirical study, artificial degradations are actualized as delays injected in RPCs. Before each session, we randomly define two recurrent artificial degradations A_1 and A_2, for example:

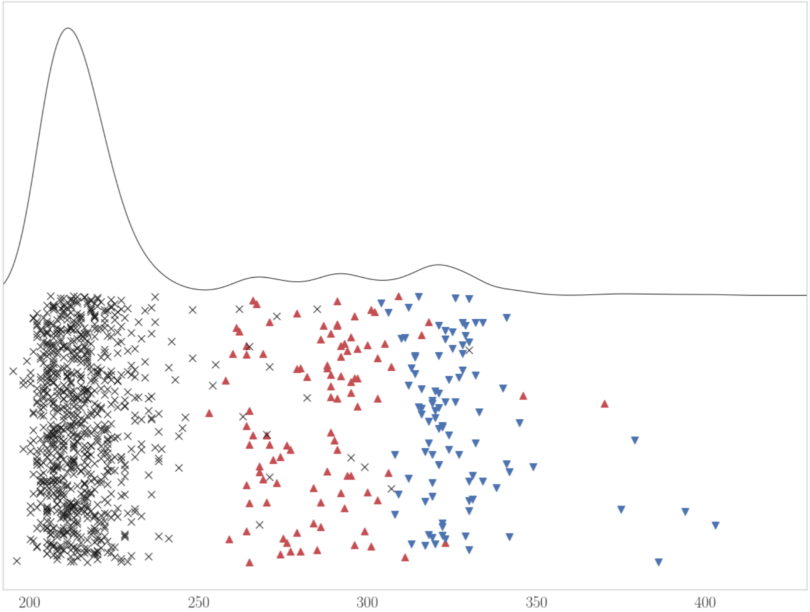


Fig. 5.5 Estimated latency distribution of requests and nominal request latencies

- A_1: 50ms delay added to `findfeaturesitems` RPC and 50 ms delay added to `getcategory` RPC;
- A_2: 50ms delay added to `gethome` RPC.

Then, during the load test, requests are randomly marked as *affected* by one of the two artificial degradations with 0.1 probability. When a request is marked as affected, the mark is propagated through RPCs, thus leveraging context propagation [84], and delays are injected in RPCs according to the artificial degradation definition. Hence, requests affected by the same artificial degradation will always have the same delays injected on the same RPCs. At the end of each load test session, each artificial degradation approximately affects 10% of requests. Figure 5.5 shows the latency behavior of the main RPC `gethome` (*i.e.,* the target of our analysis) during a load test session. Specifically, it shows the shape of the latency distribution and, under the curve, the latencies of nominal requests randomly distributed on the Y-axis. Time is expressed on X-axis in milliseconds. Requests not affected by any artificial degradation are represented as black x, the ones affected by the artificial degradation A_1 are represented as blue down triangles, whereas red up triangles represent requests affected by A_2.

Each load test session lasts 5 minutes and involves a synthetic workload simulated by Locust[8], which makes a request to the homepage every 50 milliseconds. Since not all RPCs are synchronous, some injected delays may not cause latency degradation. In order to avoid these cases, only synchronous RPCs are considered in artificial degradations. In our experiments, we consider three types of artificial degradations: *type 1* injects a delay to just one RPC, *type 2* injects delay to two RPCs and *type 3* to three RPCs. Each injected delay slows down RPC execution time by 50ms. Artificial degradations by the same type are expected to produce a similar performance degradation in terms of request latency, since the total amount of injected delays is the same.

Before each load test session, the pair of artificial degradation A_1 and A_2 is generated as follows. First, the number of RPCs affected by both artificial degradations is defined by randomly assigning *types* to A_1 and A_2, respectively. The assignment is made by ensuring that A_1 and A_2 always have different types, *i.e.,* they produce a different performance degradation in terms of request latency (see Section 5.3.4 for more details). Then, RPCs affected by delays are randomly selected, among the 6 synchronous RPCs (see Figure 5.4), for each artificial degradation according to their types.

Distributed systems are often noised, hence in order to test the robustness of our approach we also performed "noised" load testing sessions. In particular, we consider two types of noises:

- The first noise is a small deviation of delays injected in RPCs. The key insight is to reproduce situations where performance degradation doesn't show a constant behavior. For each artificial degradation, a random RPC is chosen among the affected ones so that delays injected on this RPC will not have a constant behavior, *i.e.,* the delay injected is 60ms instead of 50ms in half of the requests.

- The second type of noise involves situations where RPCs execution time degradation doesn't cause any latency degradation on the overall request. In particular, for each artificial degradation we select one of the two asynchronous calls (*i.e.,* `findfeaturesitems` and `finditems`) and inject a 100ms delay in half of requests to this call affected by artificial degradation. We have preliminarly experimented that those delays do not cause slow downs in requests.

We have performed 10 different load testing sessions with randomly generated artificial degradations, where 5 sessions are noised. This has generated 10 different sets of traces. We then ran our approach as well as baseline approaches to evaluate their effectiveness.

[8] https://locust.io/

For each load test session, we targeted a specific latency degradation interval I. We have chosen, for each session, the interval (L_{min}, L_{max}), where L_{min} and L_{max} are the minimum and the maximum observed latency of *affected* requests in the session. Our approach, as well as the one proposed by Krushevskaja and Sandler [75], requires as input a set of potential split points $\{s_0, s_1, ...s_k\}$. In each experiment, we identify the set of split points by using the same approach used in search space reduction to identify thresholds of RPC execution time (see Section 5.2.2), *i.e.,* local minima identified through Mean shift algorithm. For the considered clustering algorithms (*i.e.,* Kmeans, Hierarchical and Mean shift), we use as inputs only traces that fall in the target interval I, since the goal is to cluster degraded requests with same artificial degradations. We also set a predefined number of clusters for Kmeans and Hierarchical, we run these algorithms multiples times with different inputs (*i.e.,* k=2,...,6), and we then pick the best achieved solution for each set.

The output of each approach is a set of clusters. We select, for each injected artificial degradation, the best matching cluster for every approach, by identifying best pairs of cluster and artificial degradation $\langle C_i, A_i \rangle$, such that F1-score is maximized while considering requests affected by artificial degradation A_i as positives. At the end of this process, for each approach we have two best clusters C_1 and C_2, each one associated to the respective artificial degradation A_1 and A_2.

Finally, we then evaluate the effectiveness of each approach by using the following metrics. In theory, an ideal approach would identify clusters that correspond to the group of requests affected by same artificial degradation (*i.e.,* blue triangles and red triangles in Figure 5.5). We evaluate each approach effectiveness in terms of recall, precision and F1-score. An approach with low recall would not be adopted in practice, since it fails to detect relevant latency degradation patterns. An approach that produces results with high recall and low precision is not useful either, since identified clusters do not precisely correspond to the same artificial degradation. The three evaluation metrics are formally defined as follows. Let us name G the subset of requests that are correctly associated to the corresponding artificial degradation, P the subset of requests affected by one of the two artificial degradation, C_1 and C_2 the two identified clusters. We define the recall as $|G|/|P|$, the precision as $|G|/(|C_1| + |C_2|)$, and the F1-score as from Equation (5.2).

5.3.4 Threats to Validity

A threat to validity of our empirical study is that our experiments were performed on only one subject application, which makes it difficult to generalize the results to other distributed service-based systems. However, E-Shopper has been already used as a representative example of a microservice-based application in software performance research [4]. We expect our results to be generalizable to other distributed systems that employ a common distributed tracing solution.

Artificial delays were injected into randomly chosen RPCs at application-level. This may be a threat, since performance degradation can happen on different level of the software stack (e.g., libraries, operating systems, databases, networks, etc.), and it can be due to different reasons, such as workload spikes, network issues, contention of hardware resources and so on. However, in this work we only consider performance degradations related to RPC execution time, hence we consider injected delays as a rough but reasonable simulation of a generic performance problem (e.g. slow query, expensive computation, etc). Also, in our experiments we only consider cases where injected pairs of artificial degradations have different types. Basically, we do not consider cases where different performance issues produce same performance degradation in terms of request latency. Those cases are excluded from evaluation, because they are very specific and less frequent in practice. However, we plan to extend our study also considering these cases in future works.

A different threat is that we perform load test sessions with a single synthetic user. We used such a simple and controllable workload, because it allows us to have control on causes of relevant latency degradations. Using a more intense and mixed workload (e.g. requests to different pages of the application) may lead to more chaotic system behavior, but injected performance degradation may become not relevant and the approach difficult to evaluate. We leave the evaluation of our approach in more chaotic contexts to future works. Also, the approach is evaluated on sets of about 1000 traces, while performance debugging in modern distributed systems could involve higher number of traces. We plan to evaluate the scalability of the approach in future.

In spite of these threats, this empirical study design allowed us to evaluate our approach in a controlled setting. Thus, we are confident that the threats have been minimized and our results are reliable.

5.3.5 Baseline approaches

In this section we describe the approaches that we have used as baselines, that are: three widely popular clustering algorithms (*i.e.,* K-Means, Hierachical, Mean shift) and an F1-score-based approach [75] (that we call here KrSa). For clustering algorithms we use the implementation provided by scikit-learn Machine Learning library[9]. We have instead re-implemented the approach of Krushevskaja and Sandler, since no implementation was available, and the source code is publicly available in [32].

K-Means The K-Means algorithm [86] clusters data by trying to separate samples in k groups by equal variance, while minimizing a criterion known as within-cluster sum-of-squares. KMeans requires the number of clusters to be specified.

Hierachical Clustering (HC) Hierarchical clustering [105] is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. We use an implementation based on a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. Also hierarchical clustering requires the number of clusters to be specified.

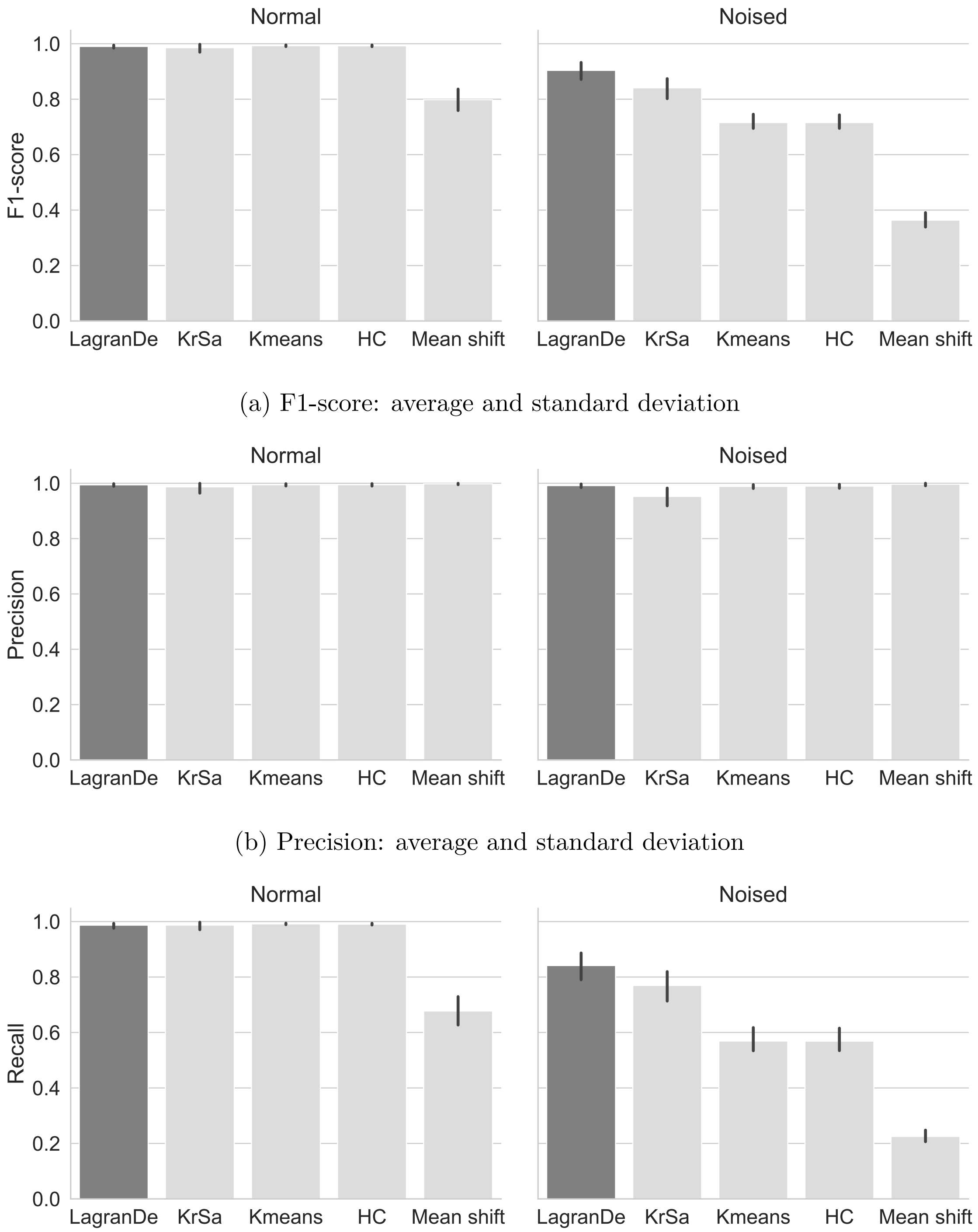
Mean shift MeanShift clustering [30] aims to discover blobs in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. The mean shift algorithm doesn't require upfront specification of number of clusters.

Krushevskaja and Sandler (KrSa). This approach [75] models the problem of detecting patterns as a binary optimization problem and uses a branch-and-bound algorithm combined with a dynamic programming algorithm to maximize the sum of the F1-scores achieved by the patterns. The approach requires the encoding of trace attributes to binary features. In our experiments, we split the RPC execution time by using the same approach used in search space reduction (*i.e.,* Mean shift algorithm, see Section 5.2.2).

5.3.6 Results

Figure 5.6a shows average and standard deviation of F1-scores achieved by each approach in both normal and noised experiments. F1-score in normal experiments suggests that all the approaches identify near optimal clusters, except for the Mean shift algorithm. Conversely, results of noised experiments show different performances among approaches. According to reported F1-scores in noised experiments (see detailed results in Table 5.2), clusters identified by LagranDe are better than those identified by machine learning clustering approaches. Kmeans, Hierarchical and Mean shift are highly precise (see Figure 5.6b), thus suggesting that each identified cluster is almost always composed by requests affected by the same artificial degradation. But their performances are low in terms of recall (see Figure 5.6c), in that an average recall of

[9] https://scikit-learn.org



(a) F1-score: average and standard deviation

(b) Precision: average and standard deviation

(c) Recall: average and standard deviation

Fig. 5.6 Results in normal and noised experiments

≈ 0.57 for Kmeans and Hierarchical, and of ≈ 0.22 for the Mean Shift algorithm have been obtained. These results suggest that identified clusters do not include significant portion of requests affected by artificial degradations. The reason of these behaviors relies on the fact that machine learning approaches blindly group together requests with similar RPC execution times, without considering their correlation with latency degradation. Therefore, performance fluctuation in RPC that doesn't produce any effect in the request, such as performance degradations in asynchronous RPC (i.e. second type of noise), can easily confuse those methods. Our approach provides on average +48% in terms of recall with respect to best clustering approaches (Kmeans and Hierarchical), without showing decrease in precision, thus leading to an improvement +26% in terms of F1-score, thereby positively answering to RQ1.

The analysis of the F1-score (see Figure 5.6a) reveals that our approach not only outperforms common machine learning approaches, but also the state-of-the-art technique against which we compare it. Our approach shows similar precision with respect to the branch and bound approach (see Figure 5.6b), but it achieves improvements in terms of recall. The result of Wilcoxon test confirms that LagranDe outperforms the approach of Krushevskaja and Sandler (KrSa) in noised experiments in terms of recall with statistical significance ($p < 0.05$) and medium effect size (Cohen's d < 0.8 and ≥ 0.5). Although both approaches are driven by the same optimization objective (*i.e.,* the sum of F1-scores), the presented technique achieves an improved recall due to the searching capability of the genetic algorithm on a wider solution space. To further confirm the improvement over the state of art, we also performed Wilcoxon test on F1-Score results. Results of statistical test confirm that LagranDe outperforms KrSa with statistical significance ($p < 0.05$) in both normal and noised experiments, where the effect size is negligible (< 0.2) in normal experiments and medium in noised experiments. These results suggest that performance analysts should use our approach for clustering requests affected by similar performance issues, since identified clusters are more comprehensives than those identified by the state of the art approach, thus positively answering to RQ2.

Both F1-score-based approaches seems to be more resilient to noise with respect to machine learning approaches. F1-score-based approaches steer the search towards clusters that are strictly related to latency degradation. Comparison of F1-score among normals and noises experiments show the robustness of our approach: LagranDe decreases its average F1-score of just $\approx 8\%$, KrSa decreases its average F1-score of $\approx 14\%$, Hierarchical and Kmeans of $\approx 28\%$, and the Meanshift of $\approx 54\%$. These results show the robustness of our approach, hence positively answering to RQ3.

	LagranDe			KrSa			Kmeans			HC			Mean shift		
ID	F1-score	Prec	Recall	F1-score	Prec	Recall	F1-score	Prec	Recall	F1-score	Prec	Recall	F1-score	Prec	Recall
0	0.844	1.0	0.73	0.743	0.741	0.745	0.678	1.0	0.513	0.678	1.0	0.513	0.349	1.0	0.211
1	0.987	1.0	0.974	0.979	1.0	0.959	0.716	1.0	0.557	0.716	1.0	0.557	0.352	1.0	0.214
2	0.855	0.986	0.754	0.714	1.0	0.555	0.716	1.0	0.558	0.716	1.0	0.558	0.341	1.0	0.205
3	0.855	1.0	0.747	0.862	1.0	0.758	0.961	0.925	1.0	0.956	0.92	0.995	0.272	1.0	0.157
4	0.992	1.0	0.984	0.769	0.765	0.773	0.722	1.0	0.565	0.722	1.0	0.565	0.272	1.0	0.158
5	0.951	1.0	0.907	0.879	1.0	0.784	0.708	0.964	0.56	0.71	0.956	0.565	0.364	1.0	0.223
6	0.992	1.0	0.984	0.888	0.799	1.0	0.681	1.0	0.516	0.681	1.0	0.516	0.43	1.0	0.274
7	0.87	1.0	0.77	0.849	1.0	0.738	0.68	0.98	0.521	0.683	0.99	0.521	0.273	1.0	0.158
8	0.997	1.0	0.995	0.847	1.0	0.734	0.696	1.0	0.534	0.692	1.0	0.529	0.317	1.0	0.188
9	0.904	0.976	0.843	0.873	0.974	0.791	0.733	1.0	0.579	0.731	0.991	0.579	0.468	1.0	0.305
10	0.837	0.993	0.724	0.871	1.0	0.771	0.696	1.0	0.534	0.696	1.0	0.534	0.353	1.0	0.215
11	0.992	1.0	0.984	0.867	1.0	0.766	0.692	0.98	0.535	0.692	0.98	0.535	0.26	1.0	0.15
12	0.984	1.0	0.969	1.0	1.0	1.0	0.982	0.965	1.0	0.982	0.965	1.0	0.374	1.0	0.23
13	0.985	0.975	0.995	0.987	1.0	0.974	0.696	1.0	0.534	0.696	1.0	0.534	0.415	1.0	0.262
14	0.815	0.951	0.714	0.619	0.978	0.453	0.706	0.939	0.565	0.706	0.939	0.565	0.395	0.923	0.251
15	0.992	1.0	0.984	1.0	1.0	1.0	0.733	1.0	0.579	0.733	1.0	0.579	0.43	1.0	0.274
16	0.926	1.0	0.862	0.676	1.0	0.51	0.7	1.0	0.538	0.696	1.0	0.533	0.259	1.0	0.149
17	0.834	1.0	0.716	0.838	1.0	0.721	0.68	1.0	0.515	0.68	1.0	0.515	0.438	1.0	0.281
18	0.828	0.953	0.732	0.663	0.695	0.634	0.7	0.964	0.549	0.7	0.964	0.549	0.429	0.981	0.275
19	0.734	1.0	0.58	0.844	1.0	0.731	0.669	0.99	0.505	0.671	1.0	0.505	0.38	1.0	0.234
20	0.997	1.0	0.995	1.0	1.0	1.0	0.683	1.0	0.518	0.683	1.0	0.518	0.398	1.0	0.249
21	0.845	1.0	0.732	0.849	1.0	0.737	0.669	1.0	0.503	0.669	1.0	0.503	0.292	1.0	0.171
22	0.878	1.0	0.782	0.875	1.0	0.777	0.689	1.0	0.526	0.689	1.0	0.526	0.407	1.0	0.255
23	0.886	1.0	0.795	0.832	1.0	0.713	0.7	0.972	0.546	0.707	1.0	0.546	0.403	1.0	0.253
24	0.853	0.993	0.747	0.855	1.0	0.747	0.691	0.99	0.531	0.691	0.99	0.531	0.318	1.0	0.189
25	1.0	1.0	1.0	0.896	0.811	1.0	0.703	1.0	0.543	0.703	1.0	0.543	0.291	1.0	0.17
26	0.852	1.0	0.742	0.852	1.0	0.742	0.688	1.0	0.524	0.688	1.0	0.524	0.418	1.0	0.265
27	0.995	1.0	0.989	0.887	0.987	0.805	0.674	1.0	0.508	0.674	1.0	0.508	0.371	1.0	0.228
28	0.955	0.923	0.99	0.73	0.829	0.653	0.714	1.0	0.555	0.709	1.0	0.55	0.585	1.0	0.414
29	0.686	1.0	0.522	0.686	1.0	0.522	0.72	1.0	0.562	0.724	1.0	0.568	0.263	1.0	0.151

Table 5.2 Detailed results of noised experiments

	Normal		Noised	
Approach	Execution time mean (sec)	Execution time standard deviation (sec)	Execution time mean (sec)	Execution time standard deviation (sec)
LagranDe	17.551	3.390	37.106	21.143
KrSa	49.070	12.820	102.259	89.291
Kmeans	0.024	0.006	0.025	0.005
HC	0.004	0.001	0.004	0.001
Mean shift	0.509	0.08	0.307	0.039

Table 5.3 Execution time in experiments: average and standard deviation

Table 5.3 shows the average execution time and standard deviation of each approach. Note that each load test session produces a set of about 1000 traces, therefore the execution time results are referred to this scale. Common clustering algorithms are extremely faster compared to F1-score-based methods (*i.e.,* LagranDe and KrSa). However they fail to achieve a good recall in noised experiments. Furthermore, both F1-score-based methods provide as output, together with each cluster, a description of its main characteristics (i.e. latency degradation pattern), which can then be used as a valuable starting point for a deeper performance analysis. Performance analyst should decide, based on the context, which technique to use. Machine learning clustering are preferable only in cases where F1-score-based methods are not able to provide solutions in a reasonable time, since the former do not provide easily interpretable results and are less robust to noise. It is important to note that efficiency of F1-score-based methods is influenced not only by the scale of the problem (number of traces under analysis) but also by the shape of RPCs execution time distributions. This aspect is suggested by the fact that execution times in both F1-score-based methods are significantly higher in noised experiments. The reason of this behavior can be explained by the fact that different shapes of RPCs execution time distribution trigger different performance behaviors in F1-score-based methods. For example, branch and bound can less frequently prune branches since bound conditions are not verified, or distributions with multiple modes can trigger a time-comsuming precomputation in our approach.

Obviously, if the analysis of requests allows to efficiently compute clusters with F1-score-based methods, our proposed approach should be preferred since it outperforms the state-of-art baseline both in terms of effectiveness and efficiency (+179% faster in normal experiments and +175% faster in noised experiments). Therefore, we positively answer also to RQ4.

For sake of completeness, all results of both normal and noised experiments are publicly available in [31].

5.4 Conclusion

In this chapter, we have introduced LagranDe, an automated approach for detecting LDPs. The approach models the detection of LDPs as an optimization problem, which is solved through a genetic algorithm and a dynamic programming algorithm. Our preliminary evaluation on a microservice-based system shows that LagranDe outperforms both clustering algorithms and the approach proposed by Krushevskaja and Sandler [75], in the defined experimental settings, especially when system runtime behavior is not very regular but it varies over time in terms of execution time of called RPCs. These results constitute a first promising step towards the design of effective automated LDPs detection techniques, since non-regular behaviors are common in live service-based systems, and make it difficult to determine, without automation, the causes of performance degradation.

Chapter 6

DeLag: Detecting Latency Degradation Patterns in Service-based Systems

This chapter presents DeLag (Detecting Latency Degradation Patterns), that is a novel automated approach for detecting LDPs. Section 6.1 describes how the problem of detecting LDPs is modeled as multi-objective optimization problem. Section 6.2 outlines the workflow used by DeLag to detect LDPs. In Section 6.3 research questions are presented along with experimental design and results. Section 6.4 discusses some implications of our findings. Section 6.5 describes threats to validity, and Section 6.6 concludes the chapter.

This work was conducted in collaboration with Vittorio Cortellessa. The content of this chapter is based on a submitted article [127], which is currently under review.

6.1 Multi-objective optimization model

Existing approaches based on F1-Score optimization, such as the one proposed by Krushevskaja and Sandler [75], and LagranDe [33], search for optimal partitions of the *targeted latency range* while considering a single optimal pattern for each sub-interval. The sum of F1-scores is maximized in order to get the optimal set of patterns. Although this technique works properly in situations where different performance issues lead to clearly distinguishable latency behaviors, its effectiveness decreases when latency degradations introduced by different issues are similar. For example, Fig. 6.1a shows a scenario with two performance issues leading to two clearly separated latency distributions. This is the ideal scenario for F1-score-based approaches, since the targeted latency range can be divided in a way that clearly separates the LDPs (e.g. pattern 2 for the $(460ms, 550ms)$ sub-interval and pattern 1 for $(550ms, 750ms)$). However, there may be cases where it is difficult to partition the targeted latency range so that patterns are clearly separated (e.g. Fig. 6.1b). This limitation was also highlighted in the work of Krushevskaja and Sandler [75] by showing that, the more latency distributions (related to different patterns) are close one to another, the more the effectiveness of the approach decreases.

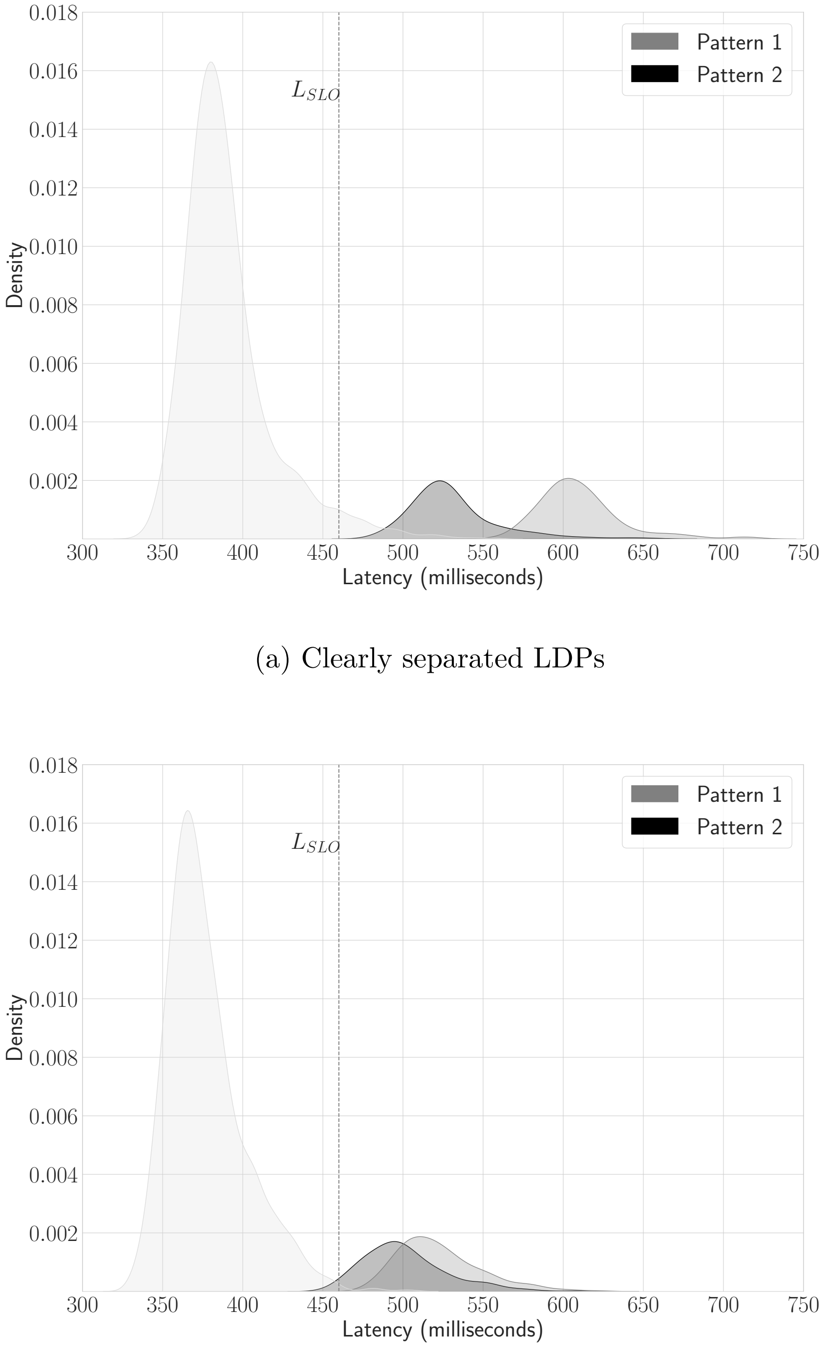
Our approach overcomes the latter problem by simultaneously searching multiple LDPs for the entire *targeted latency range*.

In this section, we describe how we model the problem of detecting LDPs as a multi-objective optimization problem. First, we define the search space of our optimization problem. Then, we describe our optimization objectives.

6.1.1 Search Space

DeLag simultaneously searches multiple patterns for the entire *targeted latency range*, therefore each possible set of patterns $S = \{P_1, P_2, ..., P_n\}$ is considered as a *solution*. As described in Chapter 4, a pattern is a set of predicates $P = \{p_1, p_2, ..., p_m\}$ and each predicate is a triple $p = \langle j, e_{min}, e_{max} \rangle$ where $[e_{min}, e_{max})$ defines the execution time range for the RPC j. RPC execution time is a continuous value, thus e_{min} and e_{max} can assume a wide range of possible values. In order to exclude, from our search space, solutions with near-similar predicates as well as unrelevant predicates (*i.e.,* related to rare execution time behaviors), we identify (through clustering method), for each RPC j, a set of eligible values E_j. Therefore, each predicate $p = \langle j, e_{min}, e_{max} \rangle$ in the solution space must be such that $e_{min} \in E_j$ and $e_{max} \in E_j$. E_j, for a given RPC j, is defined by selecting values in the RPC execution time range that separate dense regions of the execution time distribution. For example, a plausible set for the execution time of the RPC *Auth*, showed in Fig. 6.2b, could be $E_{Auth} = \{25, 175, 250, 350\}$.

The key intuition of this search space reduction is that it allows to consider only patterns related to relevant RPC execution time behaviors, while excluding from the search space those patterns related to rare transient execution time behaviors, as well as patterns that are similar in terms of RPC execution time behavior.



(a) Clearly separated LDPs

(b) Overlapping LDPs

Fig. 6.1 Two different scenarios of request latency distribution with two LDPs

6.1.2 Optimization Objectives

DeLag optimizes *pattern sets* by simultaneously maximizing *precision* and *recall*, and by minimizing *latency dissimilarity.*

In Chapter 4, we defined precision and recall to measure the quality of a single pattern (see Equations (4.3) and (4.4)), and in the following we adapt these measures to a whole *pattern set.*

We say that a request r satisfies a *pattern set* S if at least one pattern P in S is satisfied by r:

$$r \lhd S \iff \exists P \in S, \quad r \lhd P$$

It is worth noting that a request r can satisfy multiple patterns in the set S. Nevertheless, we minimize the number of requests satisfied by multiple patterns by minimizing *latency dissimilarity*, as it will be detailed later in this section.

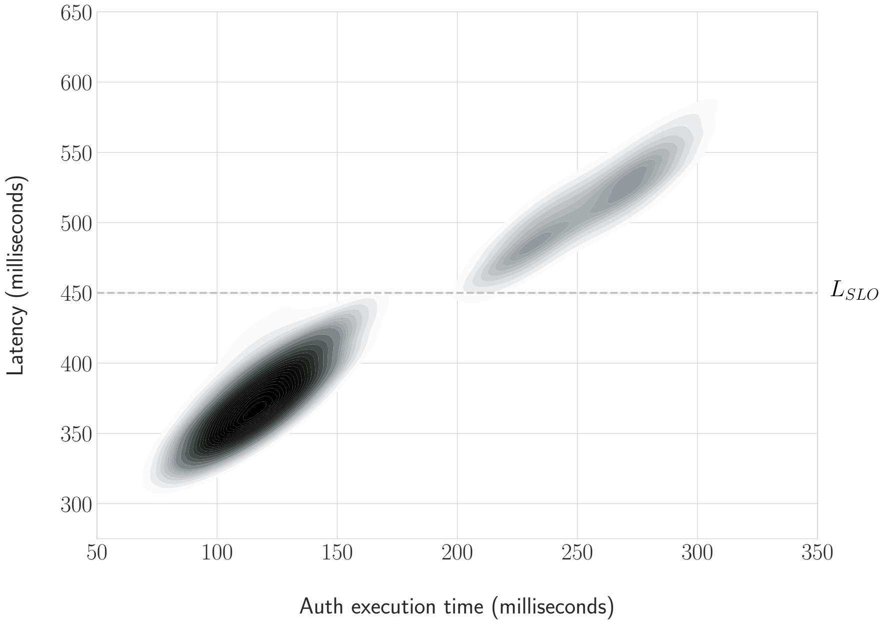
True positives and false positives for the pattern set S can be defined as follows:

$$\begin{aligned} tp &= \{r \in R_{pos} \mid r \lhd S\} \\ fp &= \{r \in R_{neg} \mid r \lhd S\} \end{aligned} \tag{6.1}$$

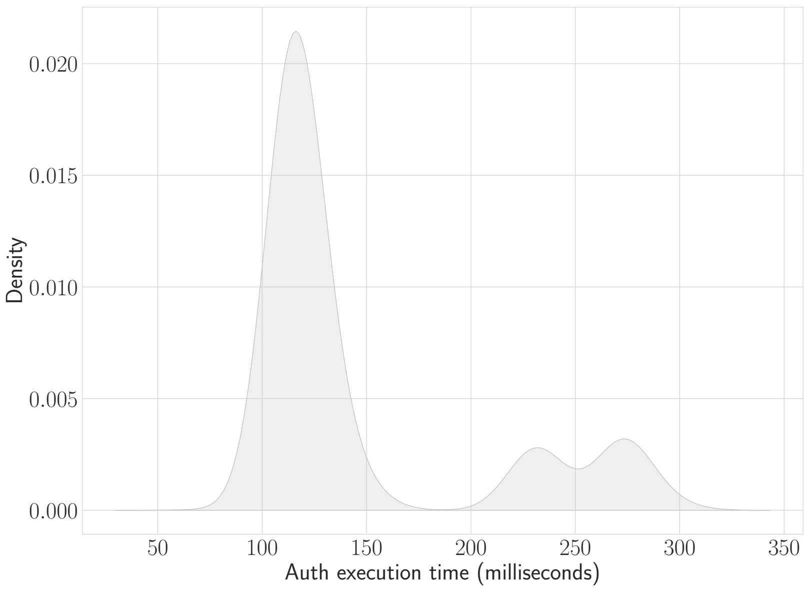
Precision measures the proportion of requests satisfied by S having latency above L_{SLO} (*i.e.,* R_{pos}), as described in Equation (4.3).

Recall, instead, measures the proportion of requests that do not meet SLO (*i.e.,* R_{pos}) and satisfy S, as described in Equation (4.4).

However, only maximizing precision and recall may not be enough. In the following we describe an exemplificative scenario, where requests affected by two distict performance issues can be satisfied by a single pattern while reaching both the maximum precision and recall. Fig. 6.2a shows the bivariate distribution of the latency for loading the homepage of a website and the execution time of an invoked RPC, namely *Auth*. Requests not meeting SLO expectation are above the L_{SLO} horizontal line. The figure shows that every request with *Auth* execution time above 175ms doesn't meet SLO expectations, therefore a solution S_1 containing a single pattern $P = \{\langle Auth, 175, 350\rangle\}$ will have both the highest possible precision and the highest possible recall. However, a closer look at the *Auth* execution time distribution (see Fig. 6.2b) shows that *Auth* anomalous executions (*i.e.,* the ones with execution time $> 175ms$) manifest two distinct behaviors, one defined by the $(175ms, 250ms)$ execution time range and the other one by $(250ms, 350ms)$. These behaviors are also reflected in the request latency distribution (see Fig. 6.2a) and may be potential symptoms of two distinct performance issues. Moreover, if *Auth* invokes others RPCs, these issues may be even rooted in dif-



(a) Bivariate distribution of request latency and Auth execution time. Darker parts of plot denote higher density.



(b) Auth execution time distribution

Fig. 6.2 Example of request latency distribution and execution time distribution of an invoked RPC (Auth).

ferent RPCs. A better solution would be $S_2 = \{P_1, P_2\}$, where $P_1 = \{\langle Auth, 175, 250\rangle\}$ and $P_2 = \{\langle Auth, 250, 350\rangle\}$. While keeping the same precision and recall, S_2 provides a more informative view on the nature of the latency degradation. Indeed, P_1 and P_2 identify two clusters of requests with different performance behaviors.
In order to avoid shallow solutions like S_1, we penalize *pattern sets* where latencies of requests within each cluster are diverse, by minimizing *latency dissimilarity*.

Latency dissimilarity is the sum of the average squared distance of latencies from the mean value, within each cluster of requests. Each pattern $P \in S$ identifies a set of requests $C_P = \{r \in R \mid r \lhd P\}$. Latency dissimilarity, for a given pattern set S, can be computed as follows:

$$\sum_{P \in S} \sum_{r \in C_P} (L_r - \mu_P)^2 \tag{6.2}$$

where L_r is the latency for the request r and μ_P is the average latency for requests satisfied by P:

$$\mu_P = \frac{\sum_{r \in C_P} L_r}{|C_P|} \tag{6.3}$$

Furthermore, the minimization of latency dissimilarity reduces the chance that the same request satisfies multiple patterns in S. Indeed, if the same request satisfies multiple patterns then latency dissimilarity tends to increase as the same request r will contribute multiple times to the summation in Equation (6.2).

Our optimization model involves three orthogonal objectives, *i.e.,* maximizing *precision* and *recall* while minimizing *latency dissimilarity*. We use Pareto optimality to plot the set of non-dominating solutions.

6.2 The DeLag Approach

DeLag workflow is depicted in Fig. 6.3. Firstly, DeLag starts by constructing the search space of the optimization problem (*Search Space Construction*). Secondly, it precomputes results of inequality checks and stores them in lookup tables that will be then used to avoid repeated computation in fitness evaluation (*Precomputation*). Thirdly, it generates as set of non-dominated Pareto-optimal *patterns sets* through a multi-objective evolutionary algorithms (*Genetic Algorithm*). Finally, it employs a heuristic to select a single *pattern set* from the set of Pareto-optimal solutions as the final solution (*Decision Maker*).

In the following we describe details of each workflow component. Section 6.2.1 describes *Search Space Construction*. Section 6.2.2 describes the main components of

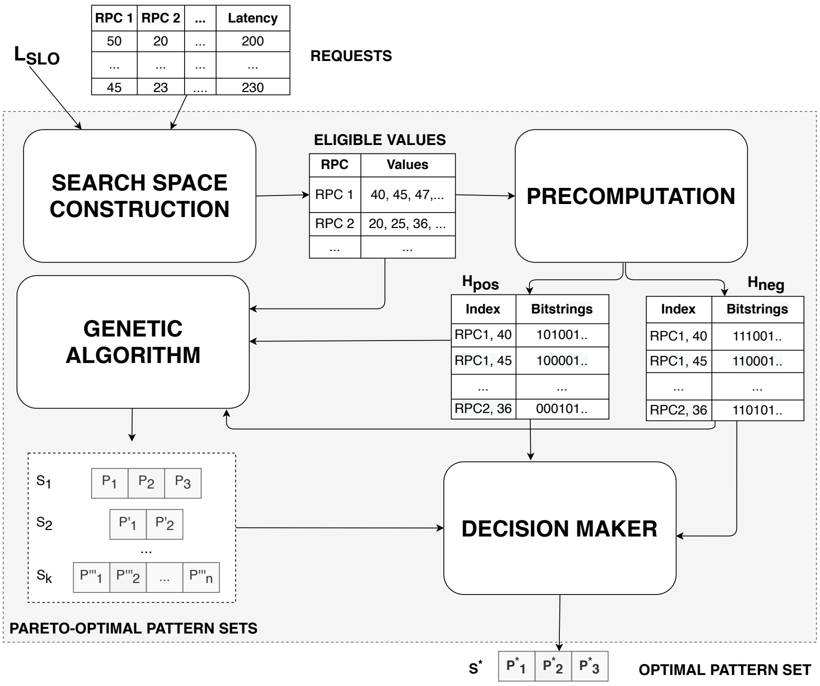


Fig. 6.3 DeLag workflow

the *Genetic Algorithm*, while Section 6.2.3 describes how *Precomputation* improves the efficiency of the evolutionary process. Finally, Section 6.2.4 outlines the *Decision Maker*.

6.2.1 Search Space Construction

The key step for shaping the search space of our problem involves the identification of highly dense regions of the RPC execution time. Basically, a set E_j of eligible values must be identified for each RPC j. As in our previous work [33], DeLag employs a Mean shift algorithm [30] to automatically identify high density intervals of the RPC execution time range. Mean shift is a feature-space analysis technique for locating maxima of a density function [26], and its application domains include cluster analysis in computer vision and image processing [30]. We use the implementation provided by *scikit-learn* [101]. For each RPC j, Mean shift algorithm clusters requests according to their corresponding execution time. We then infer split points E_j according to the highly dense identified regions. We discard clusters with size less than $|R_{pos}| \cdot 0.05$ to exclude execution time values rarely occurring in requests (further discussion on this point can be found in Section 6.5).

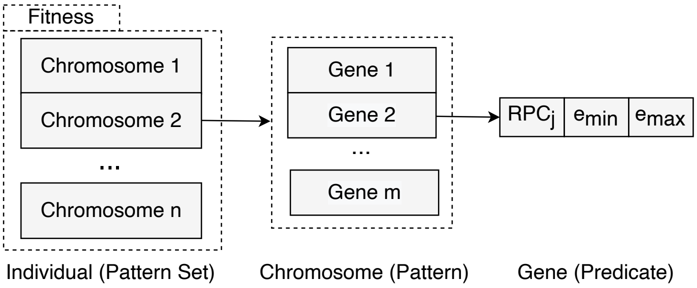


Fig. 6.4 Genetic Representation

Since eligible values selection for RPCs are independent one by another, DeLag selects E_j for each RPCs j in parallel to speed-up the process.

6.2.2 Genetic Algorithm

Algorithm 1 presents the genetic algorithm that solves the optimization problem defined in Section 6.1. DeLag uses NSGA-II [37] to build (successively-improved) Pareto-optimal solutions, while seeking new non-dominating *pattern sets*. NSGA-II is a widely-used multi-objective genetic algorithm in the context of Search Based Software Engineering [58]. The algorithm first generates a random initial population P. Then, it performs g_{max} generations while keeping track of the best individuals ever lived in the evolutionary process, namely Pareto front $\mathcal{PF}$. At each generation, a new population Q is generated by performing crossover, mutation or reproduction on randomly selected individuals. The population for the subsequent generation is then obtained by using the NSGA-II selection operator [37] on the original population P joined with the newly generated population Q. At the end of the search, the algorithm returns the set of generated solutions found to be non-dominating $\mathcal{PF}$.

The DeLag genetic algorithm is implemented on top of the *DEAP* evolutionary computation framework [48]. In the following we describe the "key ingredients" of our genetic algorithm: representation, crossover, mutation, selection and fitness.

Representation

The genetic algorithm simultaneously searches multiple LDPs, thus each individual corresponds to a whole *pattern set*. The representation of an individual is illustrated in Fig. 6.4. Our approach generates a set of of these individuals, which corresponds to a population of individuals in the evolutionary algorithm. Each individual consists of several chromosomes (patterns $\{P_1, P_2, ..., P_n\}$), and each chromosome contains

```
Algorithm 1: Genetic Algorithm
Data: max generation $g_{max}$, crossover probability $p$, mutation probability $q$
Result: Pareto front $\mathcal{PF}$
$\mathcal{PF} \leftarrow \emptyset$ ;
initialise population $\mathcal{P}$ ;
evaluate fitness of $\mathcal{P}$ and update $\mathcal{PF}$ ;
for $i$ in range(0, $g_{max}$) do
    $Q \leftarrow \emptyset$ ;
    for $j$ in range(0, $|\mathcal{P}|$) do
        $r \sim U(0, 1)$ ;
        if $r < p$ then                                        ▷ apply crossover
            randomly select $S_1$ and $S_2$ from $\mathcal{P}$ ;
            $S' \leftarrow crossover(S_1, S_2)$ ;
        else if $r < p + q$ then                               ▷ apply mutation
            randomly select $S$ from $\mathcal{P}$ ;
            $S' \leftarrow mutation(S, q)$ ;
        else                                                   ▷ apply reproduction
            randomly select $S$ from $P$ ;
            $S' \leftarrow S$ ;
        $Q \leftarrow Q \cup \{S'\}$;
    evaluate fitness of $Q$ and update $\mathcal{PF}$ ;
    $\mathcal{P}' \leftarrow sorted(Q \cup \mathcal{P}, \prec_c)$ ;    ▷ see Equation (6.4) for $\prec_c$
    $\mathcal{P} \leftarrow \mathcal{P}'[0 : |\mathcal{P}|]$ ;          ▷ new population
return $\mathcal{PF}$ ;
```

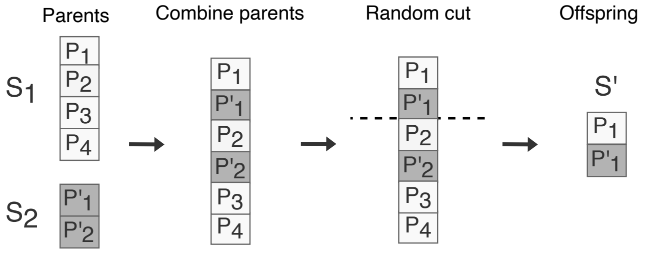


Fig. 6.5 Example of crossover operation

multiple genes (predicates $\{p_1, p_2, ..., p_m\}$), which consist of random combinations of triples $\langle j, e_{min}, e_{max}\rangle$ where $e_{min}, e_{max} \in E_j$ and j denotes the RPC j.

Crossover

The crossover operator is performed with probability p at each new generation. The operator randomly selects two individuals S_1 and S_2 that will be used to generate a new offspring individual S'. First, $S1$ and $S2$ are combined in an alternating fashion. Then, a cut point for the combined individual is randomly chosen. Finally, chromosomes at the left part of the cut point are chosen to form the new offspring individual S'. An example of a crossover operation is showed in Fig. 6.5.

Mutation

Algorithm 2 illustrates the mutation operator, which is performed with probability q at each new generation on a randomly selected individual S. Mutation is performed at two levels: individual and chromosome.

First, mutation is applied at individual level by performing one among three possible types of mutation with equal probability: *add*, *remove* or *split*. The *add mutation* randomly adds a newly generated chromosome. The *remove mutation* removes a randomly chosen chromosome from the individual. The *split mutation* splits a randomly selected chromosome P within S in two novel chromosomes P_1 and P_2. The latter operator first randomly selects a RPC j and a threshold $t \in E_j$. Then, P_1 and P_2 are created by partitioning requests that satisfy the randomly chosen P in two parts (those having $e_j < t$ and those having $e_j \geq t$). Finally, P is replaced by P_1 and P_2 in S. The detailed steps performed by the *split mutation* operator are outlined in Algorithm 3.

Then, chromosome level mutation is performed on each chromosome, in turn, with probability q. Similarly to individual level mutation, chromosome mutation applies one between two possible types of mutation with equal probability: *add* or *remove*.

Algorithm 2: Mutation

```
Data: individual $S$, mutation probability $q$
Result: mutant $S'$
$r \sim U(0, 3)$ ;
if $r < 1$ then                                        ▷ apply remove mutation (individual)
    randomly select one pattern $P$ from $S$;
    $S' \leftarrow S \setminus \{P\}$ ;
else if $r < 2$ then                                   ▷ apply add mutation (individual)
    generate a new random pattern $P$ ;
    $S' \leftarrow S \cup \{P\}$ ;
else                                                   ▷ apply split mutation (individual)
    randomly select a pattern $P$ from $S$ ;
    $P_1, P_2 \leftarrow splitPattern(P)$ ;
    $S' \leftarrow S \setminus \{P\}$ ;
    $S' \leftarrow S' \cup \{P_1, P_2\}$ ;
for each pattern $P$ in $S'$ do
    $r \sim U(0, 1)$ ;
    if $r < q$ then
        if $r < q/2$ then                              ▷ apply remove mutation (chromosome)
            randomly select one predicate $p$ from $P$;
            $P \leftarrow P \setminus \{p\}$ ;
        else                                           ▷ apply add mutation (chromosome)
            generate a new random predicate $p$ ;
            $P \leftarrow P \cup \{p\}$ ;
return $S'$ ;
```

Algorithm 3: splitPattern

```
Data: pattern $P$
Result: pattern $P_1$, pattern $P_2$
randomly select a RPC $j$ ;
select a predicate $p = \langle k, e_{min}, e_{max} \rangle$ with $k = j$ ;
if $p$ exists then
    $e'_{min} \leftarrow e_{min}$ ;
    $e'_{max} \leftarrow e_{max}$ ;
    $P' \leftarrow P \setminus \{p\}$ ;
else
    $e'_{min} \leftarrow min(E_j)$ ;
    $e'_{max} \leftarrow max(E_j)$ ;
    $P' \leftarrow P$ ;
randomly select $t \in E_j$ s.t. $e'_{min} < t < e'_{max}$ ;
$p_1 \leftarrow \langle j, e'_{min}, t \rangle$ ;
$p_2 \leftarrow \langle j, t, e'_{max} \rangle$ ;
$P_1 \leftarrow P \cup \{p_1\}$ ;
$P_2 \leftarrow P \cup \{p_2\}$ ;
return $P_1$, $P_2$ ;
```

The *add mutation* randomly adds a newly generated gene, while the *remove mutation* removes a randomly chosen gene from the chromosome.

Selection

We employed the widely used elitism method defined in NSGA-II [37]. This method defines a comparison operator $\prec_c$ based on rank and crowding-distance. The rank is a measure of level of non-domination of the individual, while the crowding-distance is a measure of density of individuals in the neighborhood.
For two pattern sets S_1 and S_2, we say $S_1 \prec_c S_2$ if and only if:

$$S_1^{rank} < S_2^{rank} \vee (S_1^{rank} = S_2^{rank} \wedge S_1^{dist} > S_2^{dist}) \tag{6.4}$$

This selection policy favors individuals with smaller non-domination rank and, when the rank is equal, it favors the one with greater crowding distance (less dense region).

Fitness

The individual fitness value is recorded as a triple: precision, recall and latency dissimilarity. In order to avoid overfitting, solutions containing patterns with recall ≤ 0.05 are penalized by assigning them the worst fitness, i.e. $\langle 0, 0, +\infty \rangle$ (further discussion on this point can be found in Section 6.5).

Fitness evaluation can be time consuming, but it is also easily parallelizable[21]. Therefore, for sake of efficiency, DeLag supports parallel fitness evaluation by assigning individuals to multiple fitness evaluators, which may run on distributed machines (a single multicore machine was used in our experimental evaluation, when comparing DeLag with other techniques).

In our experimental evaluation the initial population was created by generating 30 random individuals, crossover and mutation probability were fixed to 0.6 and 0.4, respectively, and the evolutionary process terminates after 300 generations.

6.2.3 Precomputation

The identification of true positives and false positives for a pattern set S (see Equation (4.5)) is the key operation for computing precision and recall. This operation requires to verify, for each request $r \in R$, whether $r \lhd S$. This verification involves several *inequality checks*, which are likely to be repeated several times during the evo-

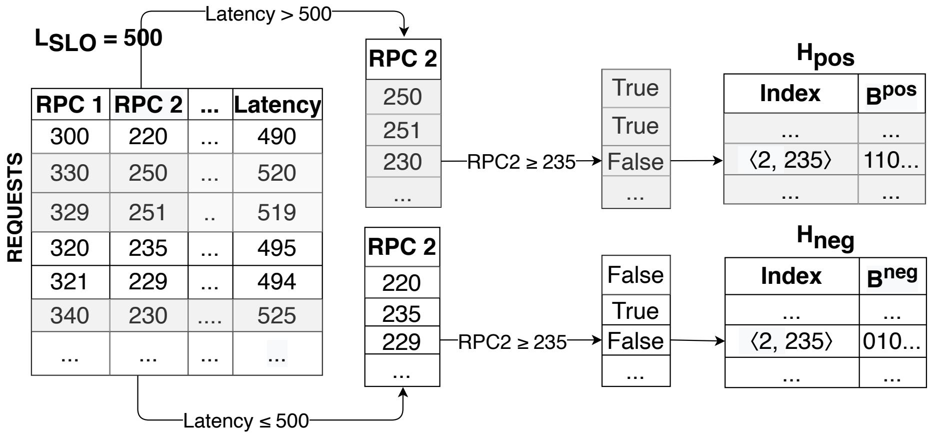


Fig. 6.6 Precomputation for $\langle j, t\rangle$ inequality check $\langle 2, 235\rangle$.

lution process. DeLag reduces the fitness evaluation effort by precomputing inequality check results, thus avoiding redundant computation. Inequality checks are denoted as pairs $\langle j, t\rangle$, where j is a RPC and $t \in E_j$ is an eligible value, hence an execution time threshold. Inequality check results are denoted as ordered sequences of booleans $B_{\langle j,t\rangle} = \langle b_0^{\langle j,t\rangle}, b_1^{\langle j,t\rangle}, ..., b_n^{\langle j,t\rangle}\rangle$, where $b_i^{\langle j,t\rangle}$ refers to the check result for the request $r_i \in R$. A check result $b_i^{\langle j,t\rangle}$ for a given inequality check $\langle j, t\rangle$ and a request $r_i = (..., e_j, ...)$ is defined as:

$$b_i^{\langle j,t\rangle} = \begin{cases} True & \text{if } e_j \geq t \\ False & \text{otherwise} \end{cases}$$

Two boolean sequences (namely $B^{pos}_{\langle j,t\rangle}$ and $B^{neg}_{\langle j,t\rangle}$) are precomputed for every pair $\langle j, t\rangle$, which represent inequality check results, respectively, for positive and negative requests (i.e., R_{pos} and R_{neg}). Boolean sequences are encoded as bitstrings and stored in two lookup tables, one for positives requests H_{pos} and another one for negative requests H_{neg}, as shown in Fig. 6.3. The length of each bitstring in H_{pos} (resp. H_{neg}) is equal to the number of requests in R_{pos} (resp. R_{neg}) . A bitstring can be retrieved by lookup tables using the corresponding index, i.e. $\langle j, t\rangle$. Fig. 6.6 shows the process used to create H_{pos} and H_{neg}.

These data structures enable fast identification of true positives and false positives across multiple requests through bitwise operations. A bitwise operation works on one or more bit strings at the level of their individual bits. We use three common bitwise operators: *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$). A predicate $p = \langle j, e_{min}, e_{max}\rangle$, with $e_{min}, e_{max} \in E_j$ (see Section 6.2.1), can be efficiently evaluated on positive requests as well as on negative requests by performing the following steps. First, boolean sequences associated to inequality checks $\langle j, e_{min} \rangle$ and $\langle j, e_{max} \rangle$ are retrieved by lookup tables H_{pos} and H_{neg}. We denote them as $B^{pos}_{\langle j,e_{min} \rangle}$ and $B^{neg}_{\langle j,e_{min} \rangle}$, and $B^{pos}_{\langle j,e_{max} \rangle}$ and $B^{neg}_{\langle j,e_{max} \rangle}$. Then, positive and negative requests that satisfy predicate p are derived through bitwise operations:

$$B^{pos}_{p} = B^{pos}_{\langle j,e_{min} \rangle} \wedge \neg B^{pos}_{\langle j,e_{max} \rangle}$$

$$B^{neg}_{p} = B^{neg}_{\langle j,e_{min} \rangle} \wedge \neg B^{neg}_{\langle j,e_{max} \rangle}$$

Hence $b^p_i \in B^{pos}_p$ (resp. $b^p_i \in B^{neg}_p$) is equal to $True$ if the request $r_i \in R_{pos}$ (resp. $r_i \in R_{neg}$) satisfies the predicate p, i.e. $r_i \lhd p$, and is equal to $False$ otherwise.
The same approach is also applied to check whether a request satisfies a pattern or not, $r_i \lhd P$:

$$B^{pos}_{P} = \bigwedge_{p \in P} B^{pos}_{p}$$

$$B^{neg}_{P} = \bigwedge_{p \in P} B^{neg}_{p}$$

And to verify whether a request satisfies the whole *pattern set*, $r_i \lhd S$:

$$B^{pos}_{S} = \bigvee_{P \in S} B^{pos}_{P}$$

$$B^{neg}_{S} = \bigvee_{P \in S} B^{neg}_{P}$$

Number of true positives and false positives for a given pattern set S are obtained by counting $True$ booleans (i.e. number of 1 in the bit string) in both B^{pos}_S and B^{neg}_S:

$$|tp| = |\{b \in B^{pos}_S \mid b = True\}|$$

$$|fp| = |\{b \in B^{neg}_S \mid b = True\}|$$

Finally, precision and recall can be derived through a simple numerical computation (see Equations (4.3) and (4.4)).

6.2.4 Decision Maker

The manual identification of a relevant single *pattern set* from the Pareto-optimal set $\mathcal{PF}$ can be complex. DeLag employs a heuristic to avoid this manual process, thus enabling full automation. The decision making heuristic uses the generalized form of the F1-score formula:

$$F_{\beta}\text{-}score = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{6.5}$$

Algorithm 4: Decision maker

```
Data: Pareto front $\mathcal{PF}$
Result: Pattern set $S^*$
$\beta \leftarrow 0.1$ ;
while $\beta \leq 1$ do
    $\mathcal{S} \leftarrow \emptyset$ ;
    select $S \in \mathcal{PF}$ with maximum $F_\beta$-score ;
    add $S$ to $\mathcal{S}$ ;
    $\beta \leftarrow \beta + 0.01$ ;
select $S^* \in \mathcal{S}$ with minimum latency dissimilarity;
return $S^*$ ;
```

The F$_\beta$-score is a generalization of the F1-score that adds a configuration parameter called beta. The F1-score uses a beta value of 1.0, which gives the same weight to both precision and recall. A beta value less than 1 gives more weight to precision and less to recall, whereas a larger beta value gives less weight to precision and more weight to recall. Maximizing precision implies the minimization of false positives, whereas maximizing recall implies the minimization of false negative.

We argue that minimization of false positives is likely to be more relevant than minimization of false negatives in the context of LDPs detection. Indeed, patterns with non-negligible amounts of false positives are likely to be less meaningful (whatever the amount of false negatives is), since they are not peculiar to requests not meeting SLO. Conversely, patterns with non-negligible amount of false negatives can still be relevant if the number of false positives is low, because they are peculiar to a portion of requests in R_{pos}, therefore they are likely to be potential symptoms of performance issues. For these reasons, the decision-making heuristic sacrifices *recall* in favor of *precision* if this implies a gain in terms of *latency dissimilarity.* Algorithm 4 outlines the decision making heuristic. The heuristic selects *pattern sets* with maximum $F_{\beta\text{-}score}$ while β ranges from 0.1 (precision is weighted 10 times as much as recall) to 1 (equally weighted). Then, it chooses among the selected solutions the one with minimum *latency dissimilarity.*

In order to speed up the process, the heuristic leverages lookup tables H_{pos} and H_{neg} generated in the precomputation phase.

6.3 Evaluation

We evaluated DeLag by performing a set of experiments aimed at answering the following research questions.

RQ$_1$ *Can DeLag* ***effectively*** *detect LDPs?* In this RQ, the LDPs detected by DeLag are compared to the ones detected by prior work and general purpose clustering algorithms. The effectiveness of the methods are compared on precision (Q_{prec}), recall (Q_{rec}) and F1-score (Q_{F1}), as they will be defined in Section 6.3.3.

RQ$_2$ *How do* ***overlapping patterns*** *affect DeLag effectiveness?* F1-score-based techniques are less effective when distinct patterns lead to partially (or entirely) overlapping latency distributions [75]. With this RQ, we want to check whether DeLag overcomes this limitation. We evaluate how proximity of latency distributions (related to distinct LDPs) affects our effectiveness. The approach is experimented on a variety of scenarios while controlling the proximity of latency distributions related to different patterns.

RQ$_3$ *How* ***non-critical RPC*** *execution time variation affects DeLag effectiveness?* Not all execution time variations of RPCs produce effect on latency (e.g. RPCs outside the critical path). With this RQ, we want to determine whether deviations of execution times on non-critical RPCs decrease the effectiveness of DeLag. The approach is experimented on a variety of scenarios while controlling the magnitude of increment of execution times on non-critical RPCs.

RQ$_4$ *What is the DeLag* ***efficiency****?* Modern service-based systems collect thousands of traces per day (or even more), therefore efficiency is a major concern for DeLag. In our last RQ, we evaluate the efficiency of DeLag on datasets by different sizes.

In order to assess the generality of DeLag, we carried out our experiments on two case studies based on open-source microservice-based systems. The first one relies on *E-Shopper*, an e-commerce web application, while the second one on *Train Ticket*, a web-based booking system. Both case studies are introduced in Section 6.3.1. In Section 6.3.2 we describe the techniques used as baselines. The methodology used for the evaluation is described in 6.3.3, followed by descriptions of experiments carried out to address each research question, respectively, in Sections 6.3.4, 6.3.5, 6.3.6, and 6.3.7.

6.3.1 Case studies

E-Shopper

The first case study is based on E-Shopper[1], a small-size microservice-based system, already used in the evaluation of our previous works [33, 4]. E-Shopper is a typical

[1] https://github.com/SEALABQualityGroup/E-Shopper

e-commerce application, which is developed as a suite of small services, each running in its own Docker[2] container. It is designed using microservice design principles. Microservices are developed in Java and interactions among them are based on RESTful APIs. The application produces execution traces that are reported and collected by Zipkin, i.e. a popular distributed tracing system [83], and stored in Elasticsearch[3]. We focus our experimentation on requests loading the homepage, which involve a number of 25 invocations of 7 unique RPCs spread across 5 microservices.

Train Ticket

The second case study is based on Train Ticket[4], which is the largest and most complex open source microservice-based system (within our knowledge at the time of writing). The system was already used in previous software engineering studies [138, 139] . Train Ticket provides typical train ticket booking functionalities such as ticket enquiry, reservation, payment, change, and user notification. It is designed using microservice design principles and covers different interaction modes such as synchronous and asynchronous invocations, and message queues. The application produces execution traces that are reported and collected by Jaeger, and stored in Elasticsearch. The system contains 41 microservices related to business logic (uses four programming languages Java, Python, Node.js, and Go) with each service running in its own Docker container. Our experimentation focuses on requests devoted to the ticket searching process, which involve a number of 48 invocations of 14 unique RPCs spread across 9 microservices.

6.3.2 Baselines techniques

We compare DeLag against both domain-specific state-of-the-art approaches and general-purpose clustering algorithms. The latters were considered because of their straightforward application to the subject problem, i.e. the identification of clusters of requests that shows similar behavior in terms of RPCs execution time. We considered two widely popular clustering algorithms, i.e. *K-Means*[86] and *Hierarchical clustering*[105], and for both of them we used the implementation provided by scikit-learn [101].

As domain-specific baselines we considered both F1-score-based methods (i.e., our previous work [33] and the one proposed by Krushevskaja and Sandler [75]), and a

[2] https://www.docker.com
[3] https://www.elastic.co/elasticsearch
[4] https://github.com/FudanSELab/train-ticket

recently proposed random-forest-based approach [9]. Unfortunately, only our previous work provides publicly available source code, therefore we have re-implemented both the approach of Krushevskaja and Sandler [75] and the one of Bansal et al.[9], which are now publicly available in the replication package [128].
In the following we provide descriptions of each baseline technique:

K-Means. The K-Means algorithm [86] clusters data by trying to separate samples in k groups, while minimizing a criterion known as within-cluster sum-of-squares. K-Means requires the number of clusters to be specified. In our experimentation, we execute the algorithm with k ranging from 2 to 10 and we pick the best solution among them.

Hierachical clustering (HC). Hierarchical clustering [105] is a general family of clustering algorithms that build nested clusters by progressively merging or splitting them. We use an implementation based on a bottom-up approach: each observation starts in its own cluster, and clusters are successively merged together. Also hierarchical clustering requires the number of clusters to be specified, therefore we employed the same approach used for the K-Means algorithm.

Krushevskaja and Sandler (KrSa). This approach[75] models the problem of detecting patterns as a binary optimization problem and uses a branch-and-bound algorithm combined with a dynamic programming algorithm to maximize the sum of the F1-scores achieved by the patterns. The approach requires the encoding of trace attributes to binary features and the selection of a set of split points to divide the targeted latency range. Similarly to what has been done here in the search space construction phase (Section 6.2.1), our re-implementation of this approach uses Mean shift algorithm [30] for both encoding and split points selection. In order to avoid overfitting, we force the Mean Shift algorithm to discard clusters of requests with size less or equal to $|R_{POS}| \cdot 0.05$. We used the implementation of this approach already used in our previous work [33].

LagranDe. Similarly to KrSa, LagranDe [33] searches for the optimal set of patterns that maximize the sum of the F1-scores. The main difference relies on the technique used to search the optimal pattern for each sub-interval, which is based on a genetic algorithm. More details on the approach can be found in Chapter 5. We used the same experimental setup used for KrSa for both encoding step and split point selection, while the size of population of the genetic algorithm is set to 30. The genetic algorithm performs 300 iterations with mutation and crossover probability set to 0.4 and 0.6, respectively. In the experiments, we used the original implementation of the approach [32].

DeCaf. DeCaf[9] trains a random forest model [18] and then infers predicates correlated with anomalous behavior. We used the implementation of the random forest algorithm based on classification trees provided by scikit-learn [101]. Predicates extraction and deduplication were re-implemented in Python. In order to avoid overfitting, the minimum number of training data in a leaf is set to $|R_{POS}| \cdot 0.05$. The number of trees and features sampling ratio are set to 50 and 0.6, respectively, as in the original paper [9]. The output of the algorithm is a ranking of predicates based on their correlation scores. In our evaluation we considered the top 10 scored predicates.

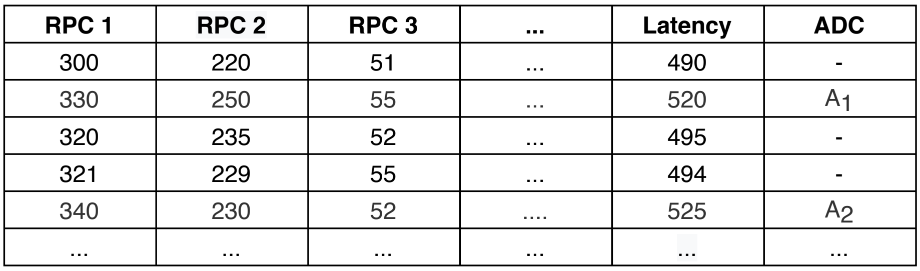
6.3.3 Methodology

In order to evaluate the effectiveness of DeLag, we run DeLag on a variety of datasets of requests containing different combinations of LDPs for both case studies. In a nutshell, each dataset used in our evaluation was generated as follows. Firstly, we altered the source code of the system to introduce delays on certain RPCs with certain probabilities, thus reproducing performance issues that lead to LDPs. Then, we performed load testing on the altered system to simulate user traffic, thus producing a dataset of requests.

Delays simulate performance issues that repeatedly occur on the system and cause latency degradation for relevant portions of requests, thus producing LDPs. We call these simulated performance issues Artificial Delay Combinations (ADCs). ADCs are designed to simulate both performance issues that are rooted in the internal implementation of individual RPCs (i.e. a single RPC is involved), and performance issues that arises from the interaction of multiple services [138] (i.e. multiple RPCs are involved). Specifically, each ADC involves from a minimum of one RPC to a maximum of three RPCs, and it is defined by a set of pairs $\langle j, d \rangle$, where d denotes the delay in milliseconds that is introduced in RPC j.

We evaluated DeLag using a variety of scenarios from both case studies, where each scenario involves two randomly generated ADCs (hence two LDPs). We developed a process that enables us to automatically alter the source code of the system by injecting delays according to the generated ADCs. Each request to the altered system is randomly assigned to one of the two ADCs with probability 0.1 (hence, with 0.8 probability is not subject to any delay) and delays are automatically introduced to the corresponding RPCs according to the ADC. For each scenario, we perform a load testing session involving 20 synthetic users simulated by Locust[5], where each user makes

[5] https://locust.io/



RPC 1	RPC 2	RPC 3	...	Latency	ADC
300	220	51	...	490	-
330	250	55	...	520	A1
320	235	52	...	495	-
321	229	55	...	494	-
340	230	52	....	525	A2
...	...	...	...	...	...

Fig. 6.7 Example of dataset

a request to the system and randomly waits 1 to 3 seconds for the next request. At the end of each session, the operational data collected by distributed tracing tools (i.e. Jaeger and Zipkin) are processed and transformed in tabular format, thus producing a dataset. Fig. 6.7 shows an example of dataset, where each row represents a request and each column contains the cumulative execution time of a RPC. It is cumulative because it represents the whole contribution, in terms of execution time, of the RPC to the whole request. In other words, if a request involves multiple invocations of the same RPC, then the cell contains the sum of all invocation execution times. The *Latency* column contains the overall request latency, while the *ADC* column reports whether the request is assigned to an ADC (A_1 or A_2) or not (-). Each dataset contains approximately 10% of requests assigned to the first ADC A_1 (we denote this subset of requests as R_{A_1}), 10% assigned to A_2 (resp. R_{A_2}) and 80% of requests showing the non-altered RPCs execution times behavior. For each scenario, L_{SLO} is defined as the smallest latency for requests assigned to one of the two ADCs.

We measure the *effectiveness* of DeLag on a given scenario (i.e. dataset) by using three quality measures: precision, recall and F1-score. DeLag outputs a *pattern set* S^*=$\{P_1, ..., P_n\}$ which identifies a set of clusters of requests $\mathcal{C}^*$=$\{C_{P_1}, ..., C_{P_n}\}$, where each cluster $C_P \in \mathcal{C}^*$ identifies the set of requests satisfied by P, i.e. C_P=$\{r \in R \mid r \lhd P\}$. In order to evaluate DeLag, we intend to verify whether there are two patterns in S^* that identify R_{A_1} and R_{A_2}, respectively. To this aim, we first identify the best matching patterns $P_{A_1}, P_{A_2} \in S^*$, where P_{A_1} (respectively P_{A_2}) is chosen by selecting the pattern that maximizes F1-score while considering requests in R_{A_1} (respectively R_{A_2}) as positives and all other requests as negatives. Once P_{A_1} and P_{A_2} are identified, precision, recall and F1-score can be derived as follows:

$$Q_{prec} = \frac{\mid C_{P_{A_1}} \cap R_{A_1} \mid + \mid C_{P_{A_2}} \cap R_{A_2} \mid}{\mid C_{P_{A_1}} \mid + \mid C_{P_{A_2}} \mid} \tag{6.6}$$

$$Q_{rec} = \frac{\mid C_{P_{A_1}} \cap R_{A_1} \mid + \mid C_{P_{A_2}} \cap R_{A_2} \mid}{\mid R_{A_1} \cup R_{A_2} \mid} \tag{6.7}$$

$$Q_{F1} = 2 \cdot \frac{Q_{prec} \cdot Q_{rec}}{Q_{prec} + Q_{rec}} \tag{6.8}$$

These measures can be also applied to evaluate baseline techniques, since F1-score-based techniques and DeCaf return set of patterns which identify clusters of requests (similarly to DeLag), while clustering algorithms directly return clusters of requests.

Both DeLag and baselines involve randomness (except for KrSa which uses a deterministic algorithm), thus to mitigate effectiveness variability we execute these techniques on each dataset 20 times.

The generation of datasets and the experimentation of techniques are performed on a dual Intel Xeon CPU E5-2650 v3 at 2.30GHz, totaling 40 cores and 80GB of RAM.

6.3.4 RQ$_1$: Effectiveness

In order to answer RQ$_1$, we generated 50 random scenarios for each case study. Note that the scenarios generated here are comparable to the "noised experiments" used in Chapter 5 (see Section 5.3.3), even though there are some differences: (i) the delays injected in the system are proportional to the average execution time of a request, (ii) there are 20 concurrent users that make requests to the system instead of a single user (*i.e.,* a more challenging workload), and (iii) the RPC execution time also takes into account the time spent in the called RPCs (*i.e.,* no "pure execution time", see Section 5.3.2). We believe that these differences improves the overall soundness of our evaluation when compared to the context adopted in Chapter 5. Additionally, based on our previous experience, we deliberately decided to not perform "normal experiments" (see Section 5.3.3) to evaluate DeLag, since they seem too trivial and potentially unrealistic. Each ADC A is randomly generated as follows. Firstly, a total delay $\mathfrak{d}$ associated with A is chosen; that is, every request assigned to A will have an overall slowdown of $\mathfrak{d}$. Secondly, 1 to 3 RPCs are randomly selected among those executed in the critical path (i.e. 5 for E-Shopper and 13 for Train Ticket). We explicitly chose RPCs in the critical-path to ensure that every delay introduced by A causes latency degradation. Thirdly, the total delay $\mathfrak{d}$ is evenly split among selected RPCs. Note that if the RPC j is called a number of $\mathfrak{i}_j$ times in the request, then the whole delay assigned to j is divided by $\mathfrak{i}_j$. At the end of this process, the ADC A is composed by a set of pairs $\langle j, d \rangle$, where d denotes the delay in milliseconds that is introduced in each execution of RPC j, and it is such that:

$$\mathfrak{d} = \sum_{\langle j,d \rangle \in A} \mathfrak{i}_j \cdot d$$

The total delay $\mathfrak{d}$ is randomly chosen in the $[L_\mu \cdot 0.2, L_\mu \cdot 0.4]$ range, where L_μ is the average request latency for the system without any ADCs. L_μ is equal to 116ms for Train ticket and to 393ms for E-Shopper; these values are derived by performing load testing sessions on the non-altered systems using the setup defined in Section 6.3.3.

Modern service-based systems involve many asynchronous interactions [138], therefore many RPCs execution times variations could not produce any degradation on request latency. In order to reproduce this behavior, we also inject a random delay $\widehat{\mathfrak{d}}$ in one non-critical RPC that doesn't produce any effect on request latency. We selected, for each case study, one asynchronous RPC whose execution time degradation doesn't cause any slowdown to requests. $\widehat{\mathfrak{d}}$ is injected in both non-altered requests and in requests assigned to ADCs (with probability 0.5). Thus 50% of requests on each scenario will manifest execution time variations on the selected non-critical RPC. Similarly to $\mathfrak{d}$, $\widehat{\mathfrak{d}}$ is randomly chosen, for each scenario, in the $[L_\mu \cdot 0.2, L_\mu \cdot 0.4]$ range. In order to ensure that $\widehat{\mathfrak{d}}$ doesn't produce any effect on request latency, we performed load testing sessions on each case study while altering each system with $\widehat{\mathfrak{d}} = L_\mu \cdot 0.4$ and $\mathfrak{d} = 0$, and we verified that the observed average latency of requests doesn't show notable deviation from L_μ.

Datasets for every scenario and case study are generated by performing load testing sessions by 20 minutes each. The generation of the 50 datasets for both case studies took ∼47 hours, while experimentation of DeLag and baselines on the generated datasets lasted ∼61 hours, which leads to an overall time of ∼4.5 days spent for RQ$_1$ experiments.

Results

For each scenario, we calculated the mean value of each quality measure, namely precision (Q_{prec}), recall (Q_{rec}) and F1-score (Q_{F1}), over 20 runs for each technique. Note that a single run is performed for KrSa, given its deterministic behavior. Fig. 6.8 shows the distribution of these values, where each boxplot contains the mean values obtained from all scenarios of a given case study for a particular technique. The higher the box plot the more effective the technique. Another aspect to take into account is how "stable" is the effectiveness provided by each technique. For example, a technique that is unstable could provide high effectiveness in certain cases and extremely low one

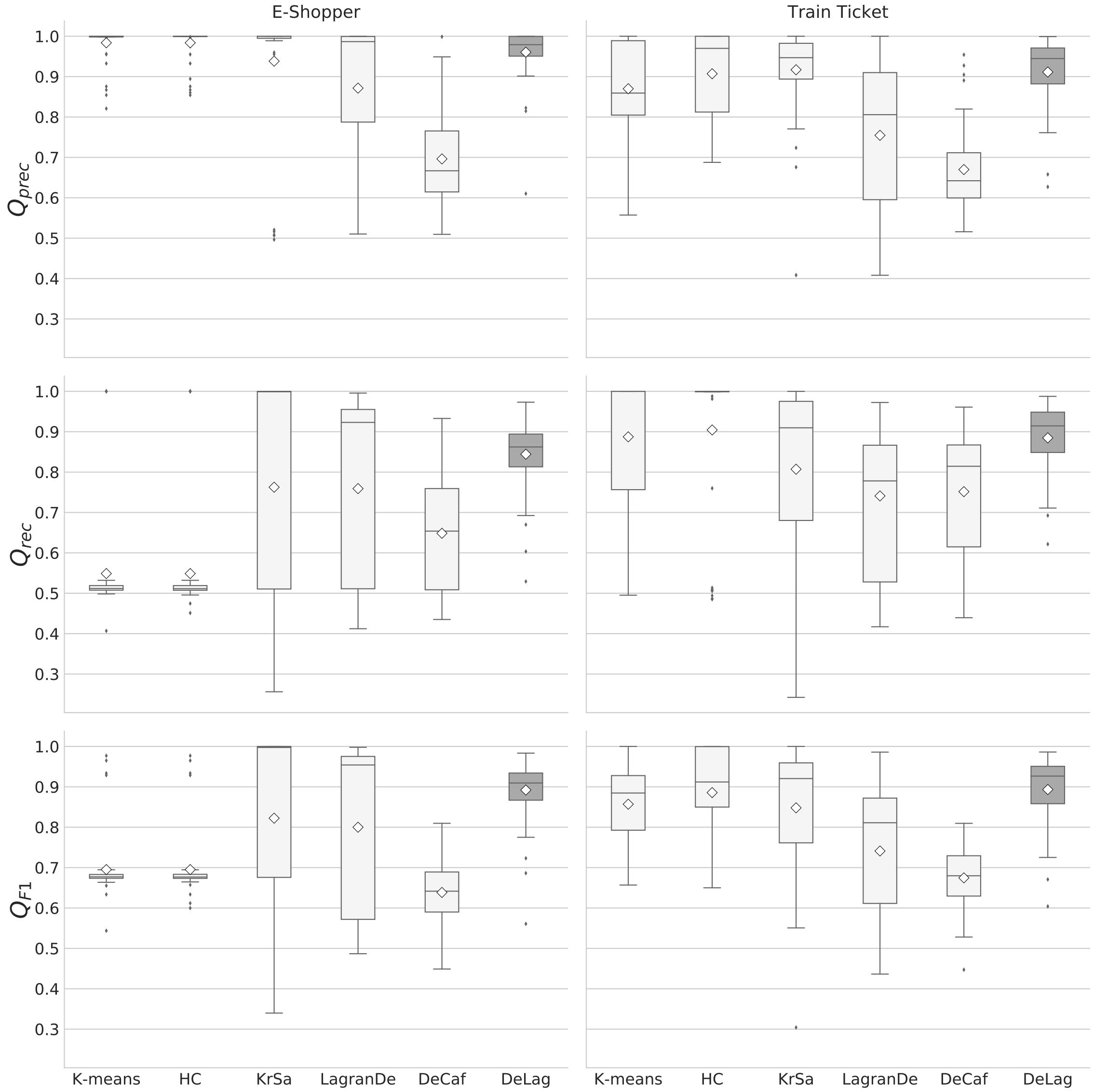


Fig. 6.8 RQ$_1$. Precision (Q_{prec}), Recall (Q_{rec}) and F1-score (Q_{F1}) for DeLag and baselines methods for each case study. The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median while the white diamond represents the mean.

Case study	Technique	Q_{prec}	Q_{rec}	Q_{F1}
E-Shopper	K-means	1.000(0.47)	<0.001(0.84)	<0.001(0.83)
	HC	1.000(0.51)	<0.001(0.84)	<0.001(0.83)
	KrSa	0.966(0.34)	0.028(0.15)	0.043(0.17)
	LagranDe	0.016(0.10)	0.034(0.06)	0.026(0.09)
	DeCaf	<0.001(0.92)	<0.001(0.75)	<0.001(0.95)
Train Ticket	K-means	0.011(0.19)	0.853(0.46)	0.002(0.22)
	HC	0.306(0.16)	0.975(0.60)	0.384(0.07)
	KrSa	0.764(0.10)	0.083(0.04)	0.186(0.03)
	LagranDe	<0.001(0.58)	<0.001(0.53)	<0.001(0.61)
	DeCaf	<0.001(0.88)	<0.001(0.59)	<0.001(0.92)

Table 6.1 RQ1. Results of the Wilcoxon test (Cliff's delta effect size in brackets) performed on the precision (Q_{prec}), recall (Q_{rec}) and F1-score (Q_{F1}), provided by DeLag compared to those provided by baseline methods.

in others. In that, a technique that provides a low variation of effectiveness is clearly desirable, since it is more reliable from a user perspective. From a bird's eye view of Fig. 6.8, we can observe that the effectiveness provided by DeLag is more "stable" compared to those provided by other techniques. The Q_{F1} first and third quartile are 0.86 and 0.93 respectively for E-Shopper, and 0.85 and 0.95 for Train Ticket, thus leading to an interquartile range (IQR) smaller than any other technique in the Train Ticket case study, and smaller than three out of the five baselines in the E-Shopper case study. The plot shows that the variation of F1-score provided by DeLag is smaller, compared to other approaches, not only within each case study, but also across them. For example, precision and recall of clustering algorithms (i.e. K-means and HC) show tiny dispersions in the E-Shopper case study, but their distributions are completely different in the Train Ticket case study, while those of DeLag just slightly change in the two case studies.

By observing Fig. 6.8, we also note that the mean F1-score (showed as white diamond) of DeLag is higher than those of baselines in both case studies. The median, instead, is greater than those of all the other baseline techniques in the Train Ticket case study, and is greater than those of 3 out of the 5 baselines in the E-Shopper case study. We report results of the Wilcoxon test (together with the corresponding Cliff's delta effect size) in Table 6.1 to compare the statistical significance and effect size of the improvements in terms of precision, recall and F1-score over the baselines. In the E-Shopper case study, DeLag outperforms ($p < 0.05$) all the other techniques in terms of F1-score (3 of them with large effect size, one with small effect size and another one with negligible effect size), while, in the Train-ticket case study, it outperforms ($p < 0.05$) 3 out of 5 baseline methods (2 with large effect size and one with small effect size). However, even in case where comparison leads to $p > 0.05$ or where effect sizes

$\mathfrak{d}_1$	$\mathfrak{d}_2$	*Distance*
$L_\mu \cdot 0.3$	$L_\mu \cdot 0.3$	$L_\mu \cdot 0$
$L_\mu \cdot 0.275$	$L_\mu \cdot 0.325$	$L_\mu \cdot 0.05$
$L_\mu \cdot 0.25$	$L_\mu \cdot 0.35$	$L_\mu \cdot 0.1$
$L_\mu \cdot 0.225$	$L_\mu \cdot 0.375$	$L_\mu \cdot 0.15$
$L_\mu \cdot 0.2$	$L_\mu \cdot 0.4$	$L_\mu \cdot 0.2$

Table 6.2 RQ$_2$. Experimental setups. Each row represents a particular setup, where $\mathfrak{d}_1$ and $\mathfrak{d}_2$ denote total delays introduced by A_1 and A_2 respectively, and *Distance* denotes expected distance between average request latency of R_{A_1} and the one of R_{A_2}.

are small or negligible, DeLag is preferable, since it provides more stable effectiveness both within the same case study and across them. For example, box plots for KrSa, whose F1-score comparison with DeLag reports $p > 0.05$ in the Train Ticket case study, show higher variability with respect to DeLag, especially in terms of recall. Another example is the comparison with HC, since it provides a similar effectiveness to DeLag in the Train Ticket case study, but F1-scores provided by HC are clearly worse (Q_{F1} <0.7) than those provided by DeLag for most of the E-Shopper scenarios.

Summing up, we answer RQ$_1$ as follows: DeLag can effectively detect LDPs, since it provides a (very often) improved and always more stable effectiveness compared to those of baseline techniques.

6.3.5 RQ$_2$: Overlapping Patterns

In order to answer RQ$_2$, we generated datasets (for both case studies), while varying the distance between the total delay introduced by A_1 and the one introduced by A_2. We defined 5 different experimental setups of delays assigned to A_1 and A_2 respectively. Table 6.2 reports these setups, which range from scenarios where latency distributions related to ADCs completely overlap (i.e. $Distance = L_\mu \cdot 0$) to scenarios where distributions are clearly separated (i.e. $Distance = L_\mu \cdot 0.2$). For each setup, we generated 20 scenarios (i.e. datasets), where total delays introduced by A_1 and A_2 are fixed by the setup, but RPCs involved in each scenario are different. Then, we avoid influence on effectiveness due to non-critical RPC execution time variation by fixing $\widehat{\mathfrak{d}}$ to $L_\mu \cdot 0.3$ in all the generated scenarios. In order to avoid extremely long experimentation time, we decreased the duration of load testing sessions to 5 minutes. Nevertheless, we expect that this does not affect the validity of our results, as the number of requests involved in each dataset is still relevant (more than 2.5k requests), and dataset size mainly affects efficiency of techniques (see RQ$_4$) rather than their effectiveness. Overall, the duration for experiments related to RQ$_2$ took ∼5 days. The dataset generation process lasted ∼43 hours, while the execution of DeLag and baseline techniques on the 200 generated datasets took ∼74 hours.

Results

We calculated the mean F1-score (Q_{F1}) among 20 runs for each technique (except for KrSa). Fig. 6.9 depicts the distributions of these means for DeLag, LagranDe and KrSa under different experimental setups. For completeness, results for other techniques (K-means, HC and DeCaf) are reported in our online appendix [128]. Fig. 6.9 shows that both KrSa and LagranDe are less effective as far as the distance between total delays introduced by A_1 and A_2 decreases (i.e., from right to left on the x-axis). The mean, the median, the first and the third quartile for KrSa and LagranDe always decrease starting from $Distance = L_\mu \cdot 0.10$ until $Distance = L_\mu \cdot 0.0$. This confirms the evidence provided by [75] about the inadequacy of F1-score-based approaches on patterns leading to similar latency behaviors. The same behavior cannot be observed on DeLag, which instead seems to improve for the same range of setups in Train Ticket and doesn't show a relevant decrease in those in E-Shopper. From Fig. 6.9 we can assert that the F1-score provided by DeLag is more stable across different setups than those of F1-score-based methods. This finding is further confirmed by Fig. 6.10, which shows mean F1-scores provided on each scenario as function of the distance between the observed average latency of requests in R_{A_1} and the one of those in R_{A_2}. Logarithmic regression lines in plots clearly show a trend towards lower effectiveness for both KrSa and LagranDe when distance between latency related to different patterns decreases. The same trend does not show up in DeLag.

Summing up, we answer RQ$_2$ as follows: closeness of latency distributions related to different patterns does not affect the effectiveness of DeLag, therefore our approach overcomes this limitation of F1-score-based methods.

6.3.6 RQ$_3$: Non-critical RPCs

Here, we intend to evaluate whether the effectiveness of DeLag is affected by execution time variations on non-critical RPCs, i.e. RPCs whose execution time variations do not cause latency degradation. Basically, they can be considered as background noise. We generated datasets for different scenarios while controlling the magnitude of delay introduced on these RPCs. Similarly to Section 6.3.4, one asynchronous RPC is selected for each case study. We used different experimental setups, where each setup is defined by a different value assigned to $\widehat{\mathfrak{d}}$, that is the amount of delay

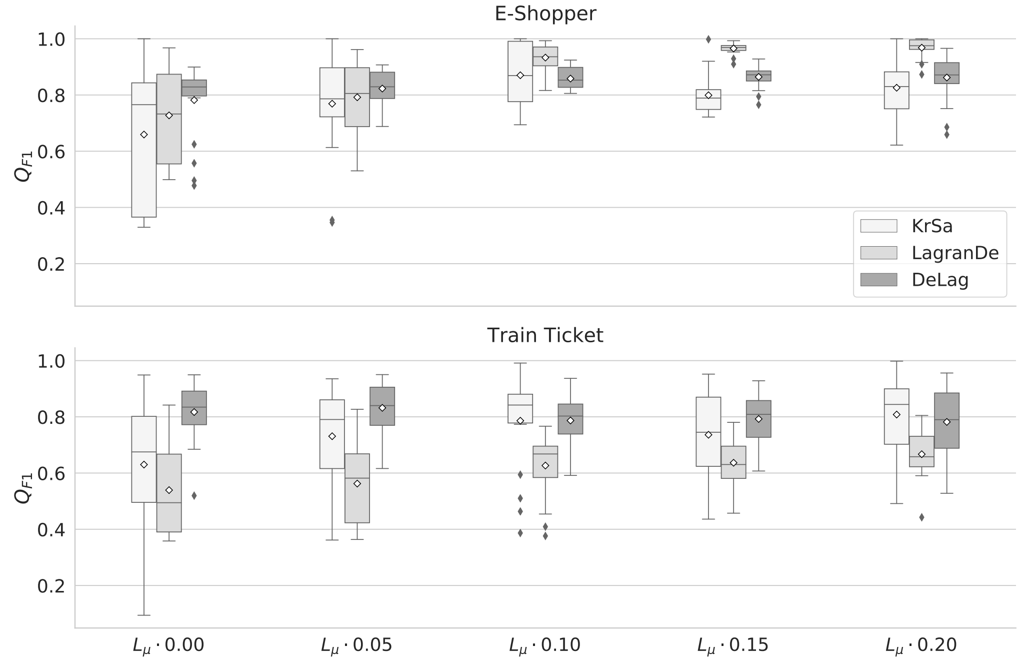


Fig. 6.9 RQ$_2$. F1-scores (Q_{F1}) for KrSa, LagranDe and DeLag under different experimental setups (see Table 6.2). The x-axis labels report the expected distance between the average latency of requests in R_{A_1} and the one of those in R_{A_2}.

introduced in the non-critical RPC. We used 5 different experimental setup that are defined by the following values of $\widehat{\mathfrak{d}}$: $L_\mu \cdot 0.0$, $L_\mu \cdot 0.1$, $L_\mu \cdot 0.2$, $L_\mu \cdot 0.3$ and $L_\mu \cdot 0.4$. For each setup we generated 20 different scenario. A delay of $\widehat{\mathfrak{d}}$ is introduced in the non-critical RPC with 0.5 probability on each request performed to the altered system. Complementarily to what we have done for RQ$_2$, in order to avoid influence on the effectiveness due to closeness of latency distributions related to distinct ADCs, here we fix delays introduced by A_1 and A_2 to $L_\mu \cdot 0.25$ and $L_\mu \cdot 0.35$ respectively. Datasets are generated by performing load testing sessions of 5 minutes, as done for RQ$_2$.

Overall, the duration for experiments related to RQ$_3$ took ~5 days. The dataset generation process lasted ~44 hours. The execution of DeLag and baseline techniques on the 200 generated datasets took ~74 hours.

Results

Fig. 6.11 shows distributions for F1-scores provided by K-Means, HC and DeLag for each experimental setup. For completeness, results for other techniques (KrSa, LagranDe and DeCaf) are reported in our online appendix [128]. Fig. 6.11 does not suggest a correlation between $\widehat{\mathfrak{d}}$ and DeLag effectiveness. For example, in the E-Shopper

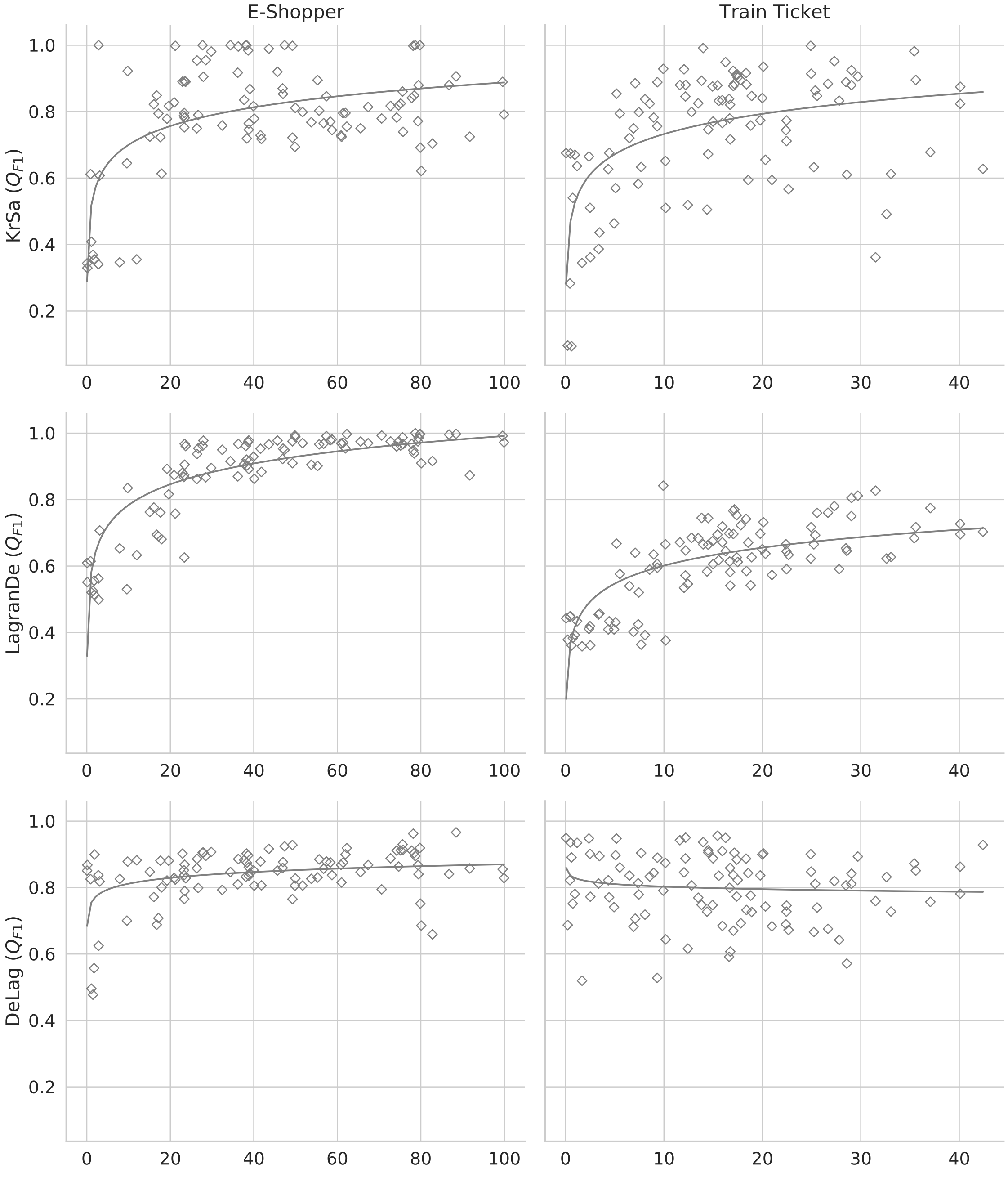


Fig. 6.10 RQ$_2$. F1-scores provided by KrSa, LagranDe and DeLag as function of the distance (in milliseconds) between the observed average latency of requests in R_{A_1} and the one of those in R_{A_2}. Each point of the plot represents the mean F1-score for the method on a particular scenario.

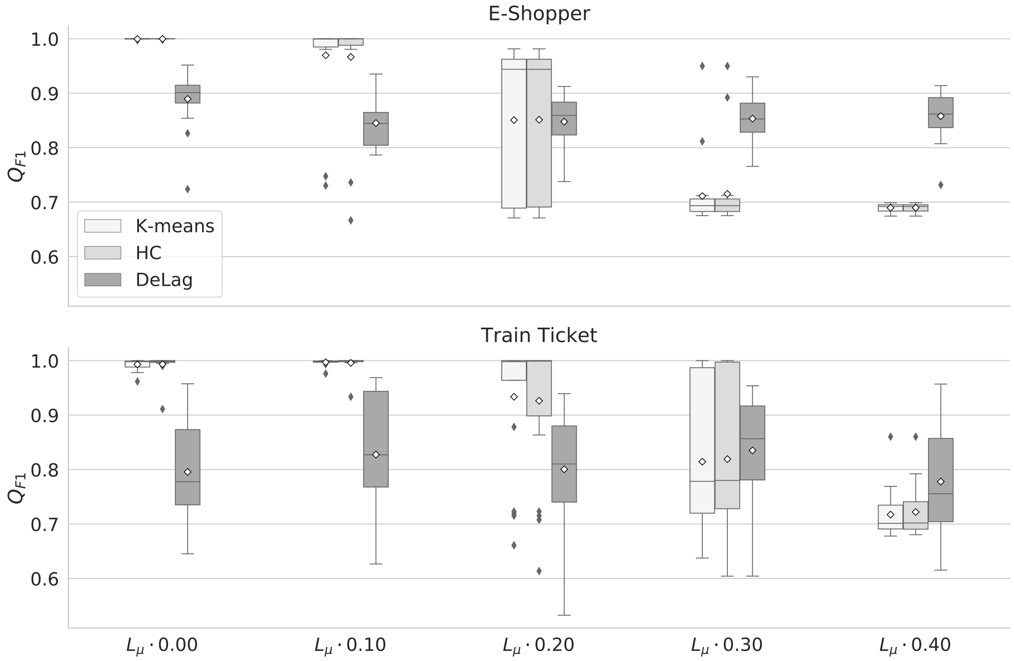


Fig. 6.11 RQ$_3$. F1-scores (Q_{F1}) for K-means, HC and DeLag under different experimental setups. The x-axis labels report the amount of delay introduced in non-critical RPCs ($\widehat{\mathfrak{d}}$) for each experimental setup.

case study, both median and mean of F1-scores decrease from scenarios with $\widehat{\mathfrak{d}} = L_\mu \cdot 0.0$ to scenarios with $\widehat{\mathfrak{d}} = L_\mu \cdot 0.1$, but they both slightly increase in all the subsequents setups. In the Train Ticket case study, instead, both mean and median of F1-scores show an alternating behavior when $\widehat{\mathfrak{d}}$ increases. On the contrary, the effectiveness of some other techniques seems to be monotonically affected by execution time variation on non-critical RPCs. For example, both K-Means and HC provide near-optimal effectiveness when $\widehat{\mathfrak{d}} = L_\mu \cdot 0.0$, i.e. there is no execution time variation on non-critical RPC, but their F1-scores significantly decrease as far as $\widehat{\mathfrak{d}}$ increases. Summing up, we answer RQ$_3$ as follows: the effectiveness of DeLag is not monotonically affected by execution time variations in non-critical RPCs. This is a fundamental step towards the adoption of DeLag in real world settings, since asynchronous interactions are pervasive in today's service-based systems.

6.3.7 RQ$_4$: Efficiency

In order to analyze the efficiency of DeLag and baseline approaches, we record the elapsed time to complete the entire end-to-end pattern detection process on different

Load testing duration (min)	Avg n° requests (E-Shopper)	Avg n° requests (Train ticket)
10	4932	5578
20	9945	11177
40	19981	22563
80	40066	45175
160	80212	90441

Table 6.3 RQ_4. Experimental setups. The first column reports duration in minutes of load testing sessions to generate each dataset. The second and the third columns report the average number of requests contained in each dataset of each setup within the same case study.

Case study	n° req.	K-means	HC	KrSa	LagranDe	DeCaf	DeLag
E-Shopper	∼4.9k	1.4	1.3	9.2	7.2	0.4	17.6
	∼9.9k	2.2	2.0	17.8	15.5	0.6	34.1
	∼20k	2.6	6.8	58.0	57.5	1.0	88.3
	∼40.1k	3.1	25.3	351.0	349.2	1.8	356.1
	∼80.2k	4.4	104.4	2428.5	2437.5	4.0	2117.5
Train Ticket	∼5.6k	2.5	2.0	258.5	21.7	0.5	48.5
	∼11.2k	2.7	3.4	318.3	23.8	0.7	44.0
	∼22.6k	3.5	10.1	406.5	101.7	1.4	113.9
	∼45.2k	4.7	64.7	1015.3	591.4	2.7	622.9
	∼90.4k	7.7	380.0	4325.6	4140.3	6.1	3548.5

Table 6.4 RQ4. Average execution times (in seconds) of techniques for each experimental setup under different case studies. The second column reports the approximate average number of requests involved in each experimental setup.

datasets with varying sizes. We generated datasets of different sizes for both case studies by using 5 different experimental setups, which control the duration of load testing sessions for each scenario (see Table 6.3).

Longer sessions obviously lead to higher number of requests to analyze, thus to more computationally expensive runs. For each setup we considered 20 different scenarios, which are randomly generated by using the same approach as for RQ_1. Note that datasets generated for Train Ticket are more computationally expensive than those of E-Shopper, since the number of RPCs under analysis is higher in the former case (25 unique RPCs compared to 7).

Unlike for effectiveness assessment, we don't expect that efficiency of techniques is influenced by randomness, therefore we perform a single run of each technique on each dataset to reduce the time effort.

The overall data generation process for RQ_4 took ∼10 days, while the execution of DeLag and baseline techniques took ∼4 days and a half.

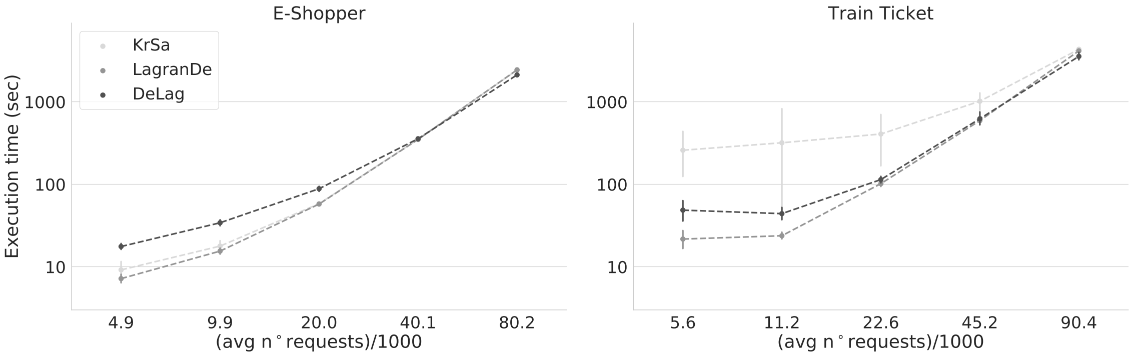


Fig. 6.12 RQ$_4$. Execution times in seconds of KrSa, LagranDe and DeLag for datasets of different sizes. Each point represent the mean execution time on 20 different scenarios within the same experiment setup, i.e. similar number of involved requests. The y axis represents the execution time, while x axis labels report the average number of requests for datasets of each experiment setup (see Table 6.3). The x and y axes are in log scale.

Results

Table 6.4 shows the average execution time of each technique for each experimental setup. DeCaf, K-means and HC severely outperform DeLag in terms of efficiency. DeCaf is 531.9 times (E-Shopper) and 580.3 times (Train ticket) more efficient than DeLag on datasets generated by 160 minutes load testing sessions, i.e. the largest datasets in our evaluation. On the same datasets, K-means and HC outperform DeLag by 479.3 and 19.3 times, respectively, in the E-Shopper case study, and by 457.4 and 8.3 times in the Train ticket case study. Despite their efficiency, these techniques have been shown to be less effective when compared to others. For example, Fig. 6.8 showed that both the mean and the median F1-score provided by DeCaf are below 0.7 in both case studies, while the mean and the median F1-score provided by clustering methods are below 0.7 in one out of the two case studies. Moreover, even KrSa and LagranDe provide a mean F1-score above 0.7 and a median F1-score above 0.8 in both case studies, therefore overall they provide better effectiveness when compared to DeCaf and general-purpose clustering algorithms.

Fig. 6.12 shows the mean execution time of techniques for each experiment setup (see Table 6.3) under each case study. On the x axes the average number of requests contained in datasets within the same experiment setup are reported. On the E-Shopper case study, both KrSa and LagranDe are more efficient than DeLag on smaller datasets. For example, KrSa and LagranDe respectively outperform DeLag by 0.92 and 1.45 times on datasets involving ∼4.9k requests. Nevertheless the figure clearly shows

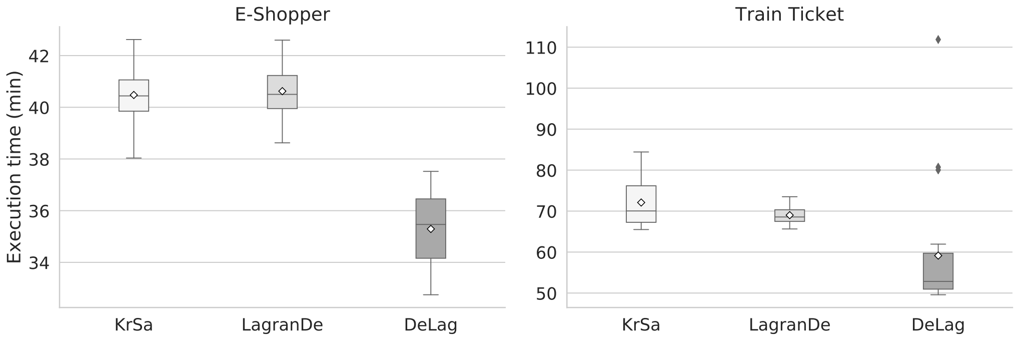


Fig. 6.13 RQ$_4$. Execution times in minutes of KrSa, LagranDe and DeLag on the largest datasets, i.e datasets generated by load testing sessions of 160 minutes.

that the difference in terms of execution times between DeLag and F1-score-based techniques decreases as the number of requests increases. Indeed, KrSa and LagranDe outperform DeLag by only 0.01 and 0.02 times, respectively, on datasets involving ∼40.1k requests, while, on the largest datasets (∼80.2 requests), the mean execution time of DeLag is smaller than those of KrSa and LagranDe.

On datasets related to Train Ticket case study, DeLag outperforms KrSa, the most effective baseline method, in all experiment setups by 0.22 to 6.23 times. When compared to LagranDe, instead, DeLag shows a similar behavior to the one observed in the E-Shopper case study; that is, LagranDe is more efficient than DeLag when dealing with smaller datasets, but as number of requests grows, the gap between the efficiency of our approach and LagranDe decreases until DeLag becomes faster than LagranDe. For example, LagranDe is more efficient than DeLag by 1.24 times on smallest datasets (∼5.6k requests). The improvement of LagranDe over DeLag becomes 0.85 on datasets with ∼11.2k requests, 0.12 on datasets with ∼22.6k requests and 0.05 on datasets with ∼45.2k requests. On the largest datasets, instead, DeLag is faster than LagranDe.

Fig. 6.13 shows execution times in minutes of each technique on the largest datasets used in our evaluation (∼80.2k requests for E-Shopper and ∼90.4k requests for Train Ticket). The mean execution time of DeLag is 35 minutes on E-Shopper and 59 minutes on Train Ticket. DeLag outperforms KrSa in efficiency by 0.15 times (E-Shopper) and by 0.22 times (Train Ticket), while LagranDe is outperformed by 0.15 times on E-Shopper and by 0.17 times on Train Ticket.

Summing up, we answer RQ$_4$ as follows: DeLag is consistently less efficient than DeCaf and general-purpose clustering algorithms. Nevertheless, these techniques provide lower effectiveness on the majority of considered scenarios when compared to DeLag. DeLag is also less efficient than the second and the third most effective technique when dealing with smaller datasets, but the efficiency of DeLag improves (when compared to these baselines) as the size of dataset increases. Moreover, DeLag clearly outperforms both KrSa and LagranDe on the largest datasets used in our evaluation.

6.4 Discussion

We found that clustering algorithms (K-means and HC) are significantly more efficient than DeLag. Nevertheless, these techniques are shown to be less effective than our approach, i.e. DeLag outperforms both clustering algorithms in one case study ($p < 0.001$ and large effect size) and provides comparable effectiveness in the other one. In addition, when compared to DeLag, these techniques show the following limitations. First, they could be ineffective on systems involving asynchronous executions of RPCs (which are nowadays pervasive), since we showed that execution time variations of non-critical RPCs severely affects their effectiveness. Second, clustering algorithms output clusters of requests without providing additional information, while DeLag also provide RPC execution time characteristics, i.e. patterns, which can be very useful for debugging purposes. We conclude that the use of DeLag is preferable over clustering algorithms.

We also found that DeLag effectiveness significantly outperforms DeCaf on both case studies (p< 0.001 with large effect size). Therefore, when effectiveness is the key priority, we suggest the use of DeLag over DeCaf. However, when higher efficiency is required and even moderate effectiveness is acceptable, DeCaf should be considered. Moreover, DeCaf capability goes beyond LDPs detection, since it enables the detection of patterns on high-ordinal categorical trace attributes (i.e categorical attributes with a high number of possible values), which is out of the scope of this work. We encourage future studies to extend the capability of our approach also to high ordinal categorical trace attributes.

DeLag also outperforms in terms of effectiveness LagranDe in the E-Shopper case study ($p < 0.001$ and large effect size), as well as in the Train Ticket case study ($p \leq 0.05$ and small effect size). When compared to KrSa, instead, DeLag provide better effectiveness in the E-Shopper case study ($p \leq 0.05$ and negligible effect size) while statistical test returns $p > 0.05$ in the Train Ticket study. Moreover, we found that effectiveness provided by F1-scores-based techniques (KrSa and LagranDe) is less stable than the one provided by DeLag (IQRs for Q_{F1} provided by DeLag are significantly smaller than those of KrSa and LagranDe). In addition, these techniques are less effective when distinct patterns lead to partially (or entirely) overlapping latency distributions, while DeLag overcomes this limitation. Our approach also outperforms in terms of efficiency F1-score-based techniques on largest datasets used in our evaluation. We conclude that the use of DeLag is preferable over F1-score-based techniques.

Overall, DeLag provides better and more stable effectiveness than other techniques. Moreover, DeLag is more efficient than the second and the third most effective techniques on the largest datasets used in our evaluation. However, when higher efficiency is required, fastest techniques such as clustering algorithms or DeCaf could be preferable. Nevertheless, practitioners have to take into account the limitations of these latter techniques.

6.5 Threats to Validity

6.5.1 Internal validity

Both DeLag and baseline techniques are subject to overfitting, i.e. patterns involving negligible numbers of requests. In order to deal with this behavior, each technique provides one or more configurable parameters. The use of different configurations may lead to different results, thus causing unfair comparison among the effectiveness provided by different techniques. On the other hand, the experimentation of techniques for different combinations of parameter values can be impractical due to extremely long execution times. In order to minimize this threat, we set parameters across different techniques with "similar policies". Namely, we used a reasonable threshold across different techniques such that each detected pattern must involve at least $|R_{pos}| \cdot 0.05$ requests (i.e. 5% of requests not meeting SLO expectations). For example, we forced the Mean Shift algorithm, used by KrSa and LagranDe for encoding and split point selection, to discard clusters of requests with sizes less or equal to $|R_{pos}| \cdot 0.05$. Similarly, the Mean Shift algorithm used in the Search Space Construction phase of DeLag is forced to discard clusters smaller than $|R_{pos}| \cdot 0.05$. The same threshold (i.e. $|R_{pos}| \cdot 0.05$) is also used in the Genetic Algorithm of DeLag to penalize solutions with patterns involving small numbers of requests. Finally, we used the same value to define the minimum number of training data in a leaf node [18] for the random-forest model of DeCaf.

DeLag and baselines use randomized algorithms, therefore each execution may potentially lead to different results. Guidelines to assess randomized techniques [6] recommend to perform a high number of repeated runs (e.g 1000 repetitions). However, using such a high number of repetition in our experiments would be unfeasible due to extremely long execution times. In order to have statistically significant results in a reasonable time, we performed 20 runs per technique.

6.5.2 Construct validity

We generated several scenarios to test the effectiveness of DeLag in LDPs detection. A potential threat to our work is that ADCs do not represent relevant causes of latency degradation within each scenario, i.e. they do not generate LDPs. In order to minimize this threat, we plot latencies distribution for each scenario and we check that requests assigned to ADCs are prevalent in requests showing degraded latency, i.e. $L > L_{SLO}$.

In order to have quantitative measures on the effectiveness of techniques, we chose, among the returned set of patterns, two patterns (P_{A_1} and P_{A_2}) that seems to be related to the targeted ADCs. Selecting different patterns may result in different effectiveness measures (Q_{prec}, Q_{rec}, Q_{F1}). One option we considered was to select them manually, but this approach has two cons: it takes significant human effort and it can leave room for subjectivity. Therefore we opted for an automated approach which selects, for each ADC A, the pattern with maximum F1-score while considering requests assigned to A as positives and all other requests as negatives. We are aware that patterns selected using this strategy can be suboptimal, and that there may be other patterns among those returned by each technique that can provide higher effectiveness, but overall we expect that our selection strategy is reasonable enough to evaluate the effectiveness of DeLag, and to compare it with the baseline.

Another threat to our work is that we evaluated DeLag only on scenarios where two LDPs are involved, therefore the effectiveness of techniques may change when considering more patterns. Nevertheless, we showed that our approach is more effective than those of baselines techniques when two distinct LDPs are involved.

6.5.3 External validity

DeLag achieves a high effectiveness in our evaluation. We cannot ensure that DeLag can achieve the same effectiveness on other datasets outside our experimental setup (e.g. real world scenarios). Nevertheless, through an evaluation on 700 randomly generated scenarios for two case studies, we showed that our approach is more effective than three state-of-the-art approaches and two general-purpose clustering algorithms. Datasets for scenarios are generated in laboratory since, at best of our knowledge, there are no publicly available datasets suitable to validate our work. Moreover, it is challenging to find industries that are willing to share their operational data. We limited our evaluation to these two systems since we were not able to identify other open-source service-based systems with non-trivial number of RPCs involved within each request. Nevertheless we evaluated DeLag on two systems with different characteristics and number of RPCs involved.

We evaluated the efficiency of DeLag on datasets of different sizes, ranging from 4.9k requests to 80.2k requests for E-Shopper and ranging from 5.6k requests to 90.4k requests for Train Ticket. LDPs detection in real world service-based systems may involve higher number of requests. Nevertheless, we showed that the efficiency of DeLag improves, when compared to the second and the third most effective technique, as the number of requests increases. Moreover, DeLag outperforms both these techniques on the largest datasets used in our evaluation.

6.6 Conclusion

This Chapter has presented DeLag, an automated approach to diagnose performance issues in service-based systems. Our approach leverages a search-based algorithm to detect patterns in RPC execution time behaviors correlated with latency degradation of requests, namely Latency Degradation Patterns. DeLag simultaneously searches multiple patterns while optimizing precision, recall and latency dissimilarity, and it uses a heuristic algorithm to select the optimal pattern set from the set of non-dominated solutions.

Through an evaluation of DeLag on 700 datasets, with different combinations of LDPs from two case study systems, we demonstrated that DeLag provides (very often) better and (always) more stable effectiveness than three state-of-the-art techniques and two general purpose clustering algorithms. We also demonstrated that, contrarily to other techniques, the effectiveness of DeLag is affected neither by the proximity of latency distributions related to different patterns, nor by execution time variations in non-critical RPCs. Finally, we demonstrated that DeLag is more efficient than the second and the third most effective baseline techniques when a high number of requests is involved.

We encourage future studies to put effort on the improvement of the efficiency of our approach, and to extend the scope of our approach to the detection of patterns in categorical trace attributes, especially those with high-cardinality [9].

The data and scripts used in our study are publicly available [128].

Chapter 7

Conclusion

The main research direction of this PhD thesis has been the investigation of software performance assurance in ASD/DevOps contexts.

Through our ethnographic study, we first showed that agile companies face several challenges in ensuring software performance. The lack of guidelines and well-established practices induces the adoption of approaches that can be obsolete and inadequate for the observed contexts, thereby leading to suboptimal management of performance assurance activities. An important concern is the definition of the proper work effort devoted to performance assessment activities. Indeed, different actors of an organization (*e.g.,* product owners and software architects) have different perceptions on the relevance of performance assessment activities. Hence, a clear and commonly shared definition of the amount of effort devoted to these activities is crucial to enable a reliable performance assurance process.

Continuous performance assessment plays a relevant role in ASD. Indeed, the late identification of performance bugs may imply time-consuming debugging, expensive reworks and potential technical debt. In that, performance testing automation becomes essential to enable continuous performance feedbacks against frequent code changes. A main challenge, in this regard, is the identification of a proper tradeoff between test frequency and accuracy. Indeed, accurate performance tests cannot be frequently executed, since they require complex production-like environments and high amount of hardware resources. On the other hand, lightweight performance tests are easier to automate and less demanding in terms of resources, but they are also less representative of the real system usage, hence less prone to discover performance bugs. More empirical studies are needed to understand to what extent lightweight performance tests (*e.g.,* microbenchmarks [76]) can mitigate the risks of major performance failures. Another relevant challenge for the automation of performance tests is the lack of a clear pass/fail verdict. Performance test results (*e.g.,* mean execution time) are usually compared to a baseline (*e.g.,* results on a previous software version) to determine a verdict, and it is often difficult to judge (in an automated way) whether the performance change is relevant or not. State-of-the-art approaches leverage change point detection [35], statistical process control techniques [97] or regression models [25] to identify significant changes from the history of performance results. Nevertheless, further studies are needed to design reliable automated verdicts for continuous performance assessment.

ASD advocates continuous refactoring to keep the system maintainable and easy to extend. Some agile methodologies emphasize refactoring as a key step of the software development process (*e.g.,* Extreme Programming and Test-Driven Development). In this thesis, we investigated the impact of refactoring on software performance. We provided a first empirical evidence on the relation between refactoring operations and software performance, by finding that a large part of refactoring operations cause statistically-significant performance changes. We also found that certain types of refactorings are more prone to induce intense performance regressions (*e.g.,* Extract Class), while others may cause radically different impacts depending on the context, by leading to either non-negligible regressions or sensible improvements (*e.g.,* Extract Method). Our findings constitute a first foundational step towards the development of techniques to predict the impact of refactoring operations on performance.

In the context of DevOps processes, in order to enable a faster software release cycle, companies often employ several independent teams that are responsible of loosely coupled independently deployable services (*e.g.,* microservices [96]). The continuous parallel streams of software changes and the complexity of these software systems make unfeasible the proactive assessment of performance in testing environments. In this context, the fast detection of performance issues in production becomes critical to avoid major performance failures. This thesis presented two automated techniques to detect potential symptoms of relevant performance issues in service-based systems. The presented techniques identifies RPCs execution time behaviors that are correlated with SLO violation, namely *Latency Degradation Patterns* (LDP). We first introduced *LagranDe*, an automated technique that searches for LDPs while optimizing F1-score. We presented a preliminary evaluation by showing that LagranDe outperforms in terms of effectiveness a state-of-the-art approach and two general purpose clustering algorithms. Then, we introduced *DeLag*, an automated LDPs detection technique based on multi-objective optimization. Through a comprehensive evaluation, we showed that DeLag provides (very often) better and (always) more stable effectiveness than LagranDe, two state-of-the-art techniques and two general purpose clustering algorithms.

In summary, this thesis has shown that the assessment of software performance in the context of ASD/DevOps is still far from being flawless. Further research is needed to improve the management of performance assessment activities, and to enable effective continuous performance assessment. Additionally, we showed that the continuous refactoring "attitude" of ASD is not neutral in terms of software performance and may cause non-negligible regressions. Our findings suggest that refactoring operations should be carefully performed in performance-critical components of systems. Hence, we envisage that the development of techniques to predict the impact on performance of refactoring operations would be beneficial to improve performance assurance in ASD.

Besides our empirical studies, we also presented two promising techniques to improve performance assessment in DevOps processes. We encourage future studies to put effort on improving the efficiency of the presented techniques and to extend their capability beyond RPC execution times, for example by enabling the detection of anomalous patterns in other kinds of operational data, such as categorical trace attributes (*e.g.,* HTTP headers) and time-series metrics (*e.g.,* CPU consumption and memory usage). Further research is also needed to improve automated diagnosis of performance issues in service-based systems.

References

[1] Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., and Kazman, R. (2020). How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, pages 1–1.

[2] Alsaqaf, W., Daneva, M., and Wieringa, R. (2017). Quality requirements in large-scale distributed agile projects – a systematic literature review. In Grünbacher, P. and Perini, A., editors, *Requirements Engineering: Foundation for Software Quality*, pages 219–234, Cham. Springer International Publishing.

[3] Alsaqaf, W., Daneva, M., and Wieringa, R. (2019). Quality requirements challenges in the context of large-scale distributed agile: An empirical study. *Information and Software Technology*, 110:39 – 55.

[4] Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., and Tucci, M. (2019). Exploiting architecture/runtime model-driven traceability for performance improvement. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 81–90.

[5] Arcelli, D., Cortellessa, V., Pompeo, D. D., Eramo, R., and Tucci, M. (2019). Exploiting architecture/runtime model-driven traceability for performance improvement. In *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*, pages 81–90. IEEE.

[6] Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA. Association for Computing Machinery.

[7] Ardelean, D., Diwan, A., and Erdman, C. (2018). Performance analysis of cloud applications. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 405–417, USA. USENIX Association.

[8] Auer, K. and Beck, K. (1996). *Lazy Optimization: Patterns for Efficient Smalltalk Programming*, pages 19–42. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Bansal, C., Renganathan, S., Asudani, A., Midy, O., and Janakiraman, M. (2019). Decaf: Diagnosing and triaging performance issues in large-scale cloud services.

[10] Bavota, G., Lucia, A. D., Marcus, A., and Oliveto, R. (2014a). Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer.

[11] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. (2014b). Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering (TSE)*, 40(7):671–694.

[12] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development.

[13] Behutiye, W., Karhapää, P., López, L., Burgués, X., Martínez-Fernández, S., Vollmer, A. M., Rodríguez, P., Franch, X., and Oivo, M. (2020a). Management of quality requirements in agile and rapid software development: A systematic mapping study. *Information and Software Technology*, 123:106225.

[14] Behutiye, W., Seppänen, P., Rodríguez, P., and Oivo, M. (2020b). Documentation of quality requirements in agile software development. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, pages 250–259, New York, NY, USA. Association for Computing Machinery.

[15] Beschastnikh, I., Liu, P., Xing, A., Wang, P., Brun, Y., and Ernst, M. D. (2020). Visualizing distributed system executions. *ACM Trans. Softw. Eng. Methodol.*, 29(2).

[16] Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies – a comprehensive introduction. *Natural Computing*, 1(1):3–52.

[17] Brauckhoff, D., Dimitropoulos, X., Wagner, A., and Salamatian, K. (2012). Anomaly extraction in backbone networks using association rules. *IEEE/ACM Trans. Netw.*, 20(6):1788–1799.

[18] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[19] Brutlag, J. (2009). Speed matters. [online] http://services.google.com/fh/files/blogs/google_delayexp.pdf.

[20] Bulej, L., Horký, V., Tuma, P., Farquet, F., and Prokopec, A. (2020). Duet benchmarking: Improving measurement accuracy in the cloud. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, pages 100–107, New York, NY, USA. Association for Computing Machinery.

[21] Cantú-Paz, E. and Goldberg, D. E. (2000). Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2):221 – 238.

[22] Chaparro, O., Bavota, G., Marcus, A., and Penta, M. D. (2014). On the impact of refactoring operations on code quality metrics. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 456–460.

[23] Chen, J., Yu, D., Hu, H., Li, Z., and Hu, H. (2019). Analyzing performance-aware code changes in software development process. In Guéhéneuc, Y., Khomh, F., and Sarro, F., editors, *Proceedings of the 27th International Conference on*

Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, pages 300–310. IEEE / ACM.

[24] Chen, M., Zheng, A. X., Lloyd, J., Jordan, M. I., and Brewer, E. (2004). Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43.

[25] Chen, T.-H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2017). Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 243–252.

[26] Cheng, Y. (1995). Mean shift, mode seeking, and clustering. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):790–799.

[27] Chih-Wei Ho, Johnson, M. J., Williams, L., and Maximilien, E. M. (2006). On agile performance requirements specification and testing. In *AGILE 2006 (AGILE'06)*, pages 6 pp.–52.

[28] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design ; Implementation - Volume 6*, OSDI'04, page 16, USA. USENIX Association.

[29] Collard, M. L., Decker, M. J., and Maletic, J. I. (2011). Lightweight transformation and fact extraction with the srcml toolkit. In *2011 IEEE 11th international working conference on source code analysis and manipulation*, pages 173–184. IEEE.

[30] Comaniciu, D. and Meer, P. (2002). Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619.

[31] Cortellessa, V. and Traini, L. (2019a). Detecting latency degradation patterns in service-based systems - experimental results. https://doi.org/10.5281/zenodo.3517993.

[32] Cortellessa, V. and Traini, L. (2019b). Detecting latency degradation patterns in service-based systems - source code. https://doi.org/10.5281/zenodo.3517110.

[33] Cortellessa, V. and Traini, L. (2020). Detecting latency degradation patterns in service-based systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, pages 161–172, New York, NY, USA. Association for Computing Machinery.

[34] Dai, H., Li, H., Chen, C. S., Shang, W., and Chen, T. (2020). Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering*, pages 1–1.

[35] Daly, D., Brown, W., Ingo, H., O'Leary, J., and Bradford, D. (2020). 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*, ICPE '20, pages 67–75, New York, NY, USA. Association for Computing Machinery.

[36] Davison, A. C. and Hinkley, D. V. (1997). *Bootstrap Methods and their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

[37] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197.

[38] Demeyer, S. (2005). Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls? In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 627–630.

[39] Ding, Z., Chen, J., and Shang, W. (2020). Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet? In Rothermel, G. and Bae, D., editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1435–1446. ACM.

[40] Dittrich, Y. (2002). Doing empirical research on software development: Finding a path between understanding, intervention, and method development. In *Social Thinking-Software Practice*, pages 243–262. The MIT Press.

[41] Duan, S., Babu, S., and Munagala, K. (2009). Fa: A system for automating failure diagnosis. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1012–1023.

[42] El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A., and Bouziane, A. (2020). A systematic literature review on automated log abstraction techniques. *Information and Software Technology*, 122:106276.

[43] Fagerström, M., Ismail, E. E., Liebel, G., Guliani, R., Larsson, F., Nordling, K., Knauss, E., and Pelliccione, P. (2016). Verdict machinery: On the need to automatically make sense of test results. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 225–234, New York, NY, USA. Association for Computing Machinery.

[44] Farshchi, M., Schneider, J., Weber, I., and Grundy, J. (2015). Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–34.

[45] Feitelson, D., Frachtenberg, E., and Beck, K. (2013). Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17.

[46] Fetterman, D. M. (2019). *Ethnography: Step-by-step*. Sage Publications.

[47] Fitzgerald, B., Stol, K., O'Sullivan, R., and O'Brien, D. (2013). Scaling agile methods to regulated environments: An industry case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 863–872.

[48] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., and Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *J. Mach. Learn. Res.*, 13(1):2171–2175.

[49] Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series. Addison-Wesley.

[50] Fowler, M. (2002). Yet another optimisation article. *IEEE Software*, 19(3):20–21.

[51] Fu, Q., Lou, J., Wang, Y., and Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158.

[52] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA. Association for Computing Machinery.

[53] Georges, A., Eeckhout, L., and Buytaert, D. (2008). Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 367–384, New York, NY, USA. Association for Computing Machinery.

[54] Giese, H., Lambers, L., and Zöllner, C. (2020). From classic to agile: experiences from more than a decade of project-based modeling education. In Guerra, E. and Iovino, L., editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 22:1–22:10. ACM.

[55] Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G., and Mueen, A. (2016). Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1573–1582, New York, NY, USA. Association for Computing Machinery.

[56] Han, S., Dang, Y., Ge, S., Zhang, D., and Xie, T. (2012). Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 145–155.

[57] Hanssen, G. K., Haugset, B., Stålhane, T., Myklebust, T., and Kulbrandstad, I. (2016). Quality assurance in scrum applied to safety critical software. In Sharp, H. and Hall, T., editors, *Agile Processes, in Software Engineering, and Extreme Programming*, pages 92–103, Cham. Springer International Publishing.

[58] Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1).

[59] Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130.

[60] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA.

[61] Inayat, I., Salim, S. S., Marczak, S., Daneva, M., and Shamshirband, S. (2015). A systematic literature review on agile requirements engineering practices and challenges. *Computers in Human Behavior*, 51:915 – 929. Computing for Human Learning, Behaviour and Collaboration in the Social and Mobile Networks Era.

[62] Jiang, Z. M. and Hassan, A. E. (2015). A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118.

[63] Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). An automated approach for abstracting execution logs to execution events. *J. Softw. Maint. Evol.*, 20(4):249–267.

[64] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. (2012). Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88.

[65] Johnson, M. J., Ho, C., Maximilien, E. M., and Williams, L. (2007). Incorporating performance testing in test-driven development. *IEEE Software*, 24(3):67–73.

[66] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., and Song, Y. J. (2017). Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 34–50, New York, NY, USA. Association for Computing Machinery.

[67] Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74, New York, NY, USA. Association for Computing Machinery.

[68] Kalibera, T. and Jones, R. (2020). Quantifying performance changes with effect size confidence intervals.

[69] Kasauli, R., Knauss, E., Horkoff, J., Liebel, G., and de Oliveira Neto, F. G. (2021). Requirements engineering challenges and practices in large-scale agile system development. *Journal of Systems and Software*, 172:110851.

[70] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

[71] Kim, G., Behr, K., and Spafford, G. (2013a). *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win.* IT Revolution Press, 1st edition.

[72] Kim, M., Sumbaly, R., and Shah, S. (2013b). Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104.

[73] Knuth, D. E. (2007). *Computer Programming as an Art*, page 1974. Association for Computing Machinery, New York, NY, USA.

[74] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[75] Krushevskaja, D. and Sandler, M. (2013). Understanding latency variations of black box services. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, pages 703–714, New York, NY, USA. Association for Computing Machinery.

[76] Laaber, C. and Leitner, P. (2018). An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 119–130, New York, NY, USA. Association for Computing Machinery.

[77] Laaber, C., Scheuner, J., and Leitner, P. (2019). Software microbenchmarking in the cloud. how bad is it really? *Empirical Softw. Engg.*, 24(4):2469–2508.

[78] Laaber, C., Würsten, S., Gall, H. C., and Leitner, P. (2020). Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. In Devanbu, P., Cohen, M. B., and Zimmermann, T., editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 989–1001. ACM.

[79] Leitner, P. and Bezemer, C.-P. (2017). An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA. Association for Computing Machinery.

[80] Lin, B., Nagy, C., Bavota, G., and Lanza, M. (2019). On the impact of refactoring operations on code naturalness. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*, pages 594–598. IEEE.

[81] Luo, Q., Nair, A., Grechanik, M., and Poshyvanyk, D. (2017). Forepost: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56.

[82] Mace, J. (2017a). End-to-End Tracing: Adoption and Use Cases. Survey, Brown University.

[83] Mace, J. (2017b). End-to-End Tracing: Adoption and Use Cases. Survey, Brown University.

[84] Mace, J. and Fonseca, R. (2018). Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 8:1–8:18, New York, NY, USA. ACM.

[85] Mace, J., Roelke, R., and Fonseca, R. (2016). Pivot tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO. USENIX Association.

[86] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif. University of California Press.

[87] Malik, H., Adams, B., and Hassan, A. E. (2010). Pinpointing the subsystems responsible for the performance deviations in a load test. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 201–210.

[88] McGrath, J. E. (1995). *Methodology Matters: Doing Research in the Behavioral and Social Sciences*, page 152–169. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[89] Meyer, H. (2009). *Manufacturing Execution Systems: Optimal Design, Planning, and Deployment.* McGraw-Hill Education, New York.

[90] Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., and Ouni, A. (2015). Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17:1–17:45.

[91] Morales, R., Saborido, R., Khomh, F., Chicano, F., and Antoniol, G. (2018). EARMO: an energy-aware refactoring approach for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, page 59.

[92] Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., and Succi, G. (2007). A case study on the impact of refactoring on quality and productivity in an agile team. In *Proceedings of the 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET 2007)*, pages 252–266. Springer.

[93] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276.

[94] Nagappan, M. and Vouk, M. A. (2010). Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117.

[95] Nair, V., Raul, A., Khanduja, S., Bahirwani, V., Shao, Q., Sellamanickam, S., Keerthi, S., Herbert, S., and Dhulipalla, S. (2015). Learning a hierarchical monitoring system for detecting and diagnosing service issues. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2029–2038, New York, NY, USA. Association for Computing Machinery.

[96] Newman, S. (2015). *Building Microservices.* O'Reilly Media, Inc., 1st edition.

[97] Nguyen, T. H., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2012). Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, page 299–310, New York, NY, USA. Association for Computing Machinery.

[98] Oaks, S. (2014). *Java Performance: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

[99] O'Hanlon, C. (2006). A Conversation with Werner Vogels. *Queue*, 4(4):14:14–14:22.

[100] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

[101] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[102] Ramesh, B., Cao, L., and Baskerville, R. L. (2010). Agile requirements engineering practices and challenges: an empirical study. *Inf. Syst. J.*, 20(5):449–480.

[103] Reichelt, D. G., Kühne, S., and Hasselbring, W. (2019). Peass: A tool for identifying performance changes at code level. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1146–1149. IEEE.

[104] Ren, S., Lai, H., Tong, W., Aminzadeh, M., Hou, X., and Lai, S. (2010). Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics*, 37(9):1487–1498.

[105] Rokach, L. and Maimon, O. (2005). *Clustering Methods*, pages 321–352. Springer US, Boston, MA.

[106] Rubin, J. and Rinard, M. (2016). The challenges of staying together while moving fast: An exploratory study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 982–993, New York, NY, USA. ACM.

[107] Rubin, K. S. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 1st edition.

[108] Sahin, C., Pollock, L., and Clause, J. (2014). How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10.

[109] Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., and Ganger, G. R. (2016). Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 401–414, New York, NY, USA. ACM.

[110] Sambasivan, R. R., Shafer, I., Mazurek, M. L., and Ganger, G. R. (2013). Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2466–2475.

[111] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. (2011). Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 43–56, USA. USENIX Association.

[112] Sandoval Alcocer, J. P., Beck, F., and Bergel, A. Performance evolution matrix: Visualizing performance variations along software versions. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 1–11. ZSCC: 0000001.

[113] Sandoval Alcocer, J. P. and Bergel, A. (2015). Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139, New York, NY, USA. Association for Computing Machinery.

[114] Sandoval Alcocer, J. P., Bergel, A., and Valente, M. T. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 37–48. Association for Computing Machinery. ZSCC: 0000024.

[115] Schermann, G., Cito, J., and Leitner, P. (2018). Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31.

[116] Sharp, H., Dittrich, Y., and de Souza, C. R. B. (2016). The role of ethnographic studies in empirical software engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804.

[117] Sheskin, D. J. (2007). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition.

[118] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

[119] Smith, C. and Williams, L. (2001). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley object technology series. Addison-Wesley.

[120] Stefan, P., Horky, V., Bulej, L., and Tuma, P. (2017). Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 401–412, New York, NY, USA. Association for Computing Machinery.

[121] Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M., and Flora, P. (2013). Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *2013 IEEE International Conference on Software Maintenance*, pages 110–119.

[122] Szóke, G., Antal, G., Nagy, C., Ferenc, R., and Gyimóthy, T. (2014). Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 95–104. IEEE.

[123] Tang, L., Li, T., and Perng, C.-S. (2011). Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 785–794, New York, NY, USA. Association for Computing Machinery.

[124] Tavares, C., Bigonha, M. A., and Figueiredo, E. (2020). Quantifying the effects of refactorings on bad smells. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES 2020).*

[125] Toffola, L. D., Pradel, M., and Gross, T. R. (2018). Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 314–326, New York, NY, USA. Association for Computing Machinery.

[126] Traini, L. (2021). Exploring performance assurance practices and challenges in agile software development: An ethnographic study. *Empirical Software Engineering (submitted).*

[127] Traini, L. and Cortellessa, V. (2020a). Delag: Detecting latency degradation patterns in service-based systems. *IEEE Transactions on Software Engineering (submitted).*

[128] Traini, L. and Cortellessa, V. (2020b). Delag: Detecting latency degradation patterns in service-based systems - replication package. https://github.com/SEALABQualityGroup/replication_delag.

[129] Traini, L., Di Pompeo, D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., Lanza, M., Oliveto, R., and Cortellessa, V. (2020a). How software refactoring impacts execution time. *ACM Transactions on Software Engineering and Methodology (submitted).*

[130] Traini, L., Pompeo, D. D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., Lanza, M., Oliveto, R., and Cortellessa, V. (2020b). How software refactoring impacts execution time - replication package. https://github.com/SEALABQualityGroup/replicationpackage_refperf.

[131] Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering (TSE)*, 35(3):347–367.

[132] Tsantalis, N., Ketkar, A., and Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering.*

[133] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 483–494.

[134] Veeraraghavan, K., Meza, J., Chou, D., Kim, W., Margulis, S., Michelson, S., Nishtala, R., Obenshain, D., Perelman, D., and Song, Y. J. (2016). 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)*, pages 635–651, Savannah, GA. USENIX Association.

[135] Verdecchia, R., Saez, R. A., Procaccianti, G., and Lago, P. (2018). Empirical evaluation of the energy impact of refactoring code smells. In *Proceedings of the 5th International Conference on Information and Communication Technology for Sustainability (ICT4S 2018)*, volume 52 of *EPiC Series in Computing*, pages 365–383.

[136] Wang, J. and Han, J. (2004). Bide: efficient mining of frequent closed sequences. In *Proceedings. 20th International Conference on Data Engineering*, pages 79–90.

[137] Woodside, M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE ’07, pages 171–187, USA. IEEE Computer Society.

[138] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, pages 1–1.

[139] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., and He, C. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 683–694, New York, NY, USA. Association for Computing Machinery.